

Roll No. 45

Exam Seat No.

VIVEKANAND EDUCATION SOCIETY'S INSTITUTE OF TECHNOLOGY

Hashu Advani Memorial Complex, Collector's Colony, R. C. Marg,
Chembur, Mumbai – 400074. Contact No. 02261532532



Since 1962

CERTIFICATE

Certified that Mr. Pushkar Prasad Sane
of FYMCA/A has
satisfactorily completed a course of the necessary experiments in
Python Programming Lab under my supervision
in the Institute of Technology in the academic year 20 23 – 2024.

Principal

Head of Department

Lab In-charge

Subject Teacher



V.E.S. Institute of Technology, Collector Colony,
Chembur, Mumbai, Maharashtra 400047
Department of M.C.A

INDEX

Sr. No.	Contents	Marks	Faculty Sign
1	To write, test, and debug Basic Python programs. 1. Add Three Numbers. 2. To Swap two No using third variable and without using third variable. 3. Calculate area of triangle. 4. To Solve Quadratic equations. 5. To use Bitwise operators. 6. To compute compound interest given all the required values. 7. To generate a random number between 0 and 100 . 8. To display a calendar for the January 2019. 9. To add two binary numbers.		
2	To implement Python programs with conditionals and loops. 1. To find all the prime numbers in the interval 0 to 100. 2. To check if the given number is Armstrong no or not. 3. To check if the given char is a vowel or consonant. 4. Write a Program to Take in the Marks of 3 Subjects and Display the Grade.		

	<ul style="list-style-type: none">5. To add two matrices.6. To convert a month name to a number of days.7. To check the validity of password input by users. Validation:<ul style="list-style-type: none">a. At least 1 letter between [a-z] and 1 letter between [A-Z].b. At least 1 number between [0-9].c. At least 1 character from [\$#@].d. Minimum length 6 characters.e. Maximum length 16 characters.8. To check if a number is palindrome or not.		
3	<p>To implement Python programs using List, String, Set and Dictionary.</p> <ul style="list-style-type: none">1. To merge two lists and find the second largest element in the list using bubble sort.2. To calculate the no of uppercase, lowercase letters and digits in a string.3. To count the occurrences of each word in a given string sentence.4. To add a key value pair to the dictionary and search and then delete the given key from the dictionary.5. Create one dictionary of 5 students with their name, address, age, class and marks of 5 subjects. Perform all the operations on the created dictionary.6. To concatenate two dictionaries and find sum of all values in the dictionary.7. To add and remove elements from set and perform all the set operations like Union, Intersection, Difference and Symmetric Difference.8. Perform different operations on Tuple.		

	9. Write a Python program to count the elements in a list until an element is a tuple.		
4	To implement programs on Python Functions and Modules. <ol style="list-style-type: none">1. To check whether string is palindrome or not using function recursion.2. To find Fibonacci series using recursion.3. To find the binary equivalent of number using recursion.4. To use lambda function on list to generate filtered list, mapped list and reduced list.5. Convert the temperature in Celsius to Fahrenheit in list using an anonymous function.6. To create modules in python and access functions of the module by importing it to another file/module. (Calculator program)		
5	To implement programs on OOP Concepts in python <ol style="list-style-type: none">1. Python Program to Create a Class and Compute the Area and the Perimeter of the Circle.2. To Implement Multiple Inheritance in python.3. To Implement a program with same method name and multiple arguments.4. To Implement Operator Overloading in python.5. Write a program which handles various exceptions in python.		
6	To implement programs on Data Structures using Python. <ol style="list-style-type: none">1. To Create, Traverse, Insert and remove data using Linked List.2. Implementation of stacks.3. Implementation of Queue.		

	4. Implementation of Dequeue.		
7	To implement GUI programming and Database Connectivity. <ol style="list-style-type: none"> 1. To Design Login Page. 2. To Design Student Information Form/Library management Form/Hospital Management Form. 3. Implement Database connectivity For Login Page i.e. Connect Login GUI with Sqlite3. 		
8	To implement Threads in Python. <ol style="list-style-type: none"> 1. To do design the program for starting the thread in python. 2. Write a program to illustrate the concept of synchronization. 3. Write a program for creating a multithreaded priority queue. 		
9	To implement the NumPy library in Python. <ol style="list-style-type: none"> 1. Creating ndarray objects using array() in NumPy. 2. Creating 2D arrays to implement Matrix Multiplication. 3. Program for Indexing and slicing in NumPy arrays. 4. To implement NumPy - Data Types . 		
10	To implement Pandas library in Python. <ol style="list-style-type: none"> 1. Write a Pandas program to create and display a one dimensional array-like object containing an array of data using Pandas module. 2. Write a Pandas program to convert a dictionary to a Pandas series. 3. Write a Pandas program to create a dataframe from a dictionary and display it. Sample data: {'X':[78,85,96,80,86], 		

	<p>'Y':[84,94,89,83,86],'Z':[86,97,96,72,83]}</p> <p>4. Write a Pandas program to aggregate the two given data frames along rows and assign all data.</p> <p>5. Write a Pandas program to merge two given dataframes with different columns.</p>		
--	--	--	--

Final Grade	Instructor Signature

Name of Student: Pushkar Sane			
Roll Number: 45		Lab Assignment Number: 1	
Title of Lab Assignment: To write, test, and debug Basic Python programs.			
DOP:		DOS:	
CO Mapped: CO1	PO Mapped: PO3, PO5, PSO1, PSO2	Faculty Signature:	Marks:

Aim: To write, test, and debug Basic Python programs.

Description: Below are descriptions of basic python modules, functions, etc.

1) print() Function:

For displaying the output to the user on the console, we use the print() function to print the specified message to the screen, or other standard output device.

Example:

```
print("Hello World")
```

2) Arithmetic operators:

Python Arithmetic Operators are used to perform mathematical operations like addition, subtraction, multiplication, and division. The different arithmetic operators are + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus), ** (Exponent).

3) Input Function:

For taking the input from the user, we use the built-in function input() to take the input. Since input() returns a string, we convert the string into numbers using the int() or float() function.

Example:

```
input("Enter your namer: ")
```

```
int(input("Enter your age: "))
```

4) Bitwise Operators:

In Python, bitwise operators are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed on each bit or corresponding pair of bits, hence the name bitwise operators. The result is then returned in decimal format. The different bitwise operators in python are & (Bitwise AND), | (Bitwise OR), ~ (Bitwise NOT), ^ (Bitwise XOR), << (Bitwise Left Shift), >> (Bitwise Right Shift).

5) Math Module

Python has a built-in module that you can use for mathematical tasks. The math module has a set of methods and constants. For importing a module we use the 'import' keyword.

Example:

```
math.factorial(9)
```



```
math.sqrt(16)
```

6) pow() function:

Python pow() function returns the result of the first parameter raised to the power of the second parameter.

Example:

```
pow(3,2)
print(pow(3,2))
```

7) Random module:

Python has a built-in module that you can use to make random numbers. The random module has various functions such as randint, choice, shuffle.

Example:

```
random.randint(3, 9)
random.choice(x)
random.shuffle(mylist)
```

8) Calendar module:

Python defines an inbuilt module calendar that handles operations related to the calendar. The calendar module allows us to output calendars like the program and provides additional useful functions related to the calendar. It has various functions such as month, calendar, weekday, etc.

Example:

```
calendar.month(2024, 3)
calendar.calendar(2018)
```

9) replace() method:

The replace() method replaces a specified phrase with another specified phrase. If nothing is specified in replace method then all occurrences of specified phrase will be replaced.

Example:

```
x = txt.replace("one", "three")
```

10) bin() function:

The bin() is an in-built function in Python that takes in integer x and returns the binary representation of x in a string format. If x is not an integer, then the _index()_ method needs to be implemented to obtain an integer as the return value instead of as a "TypeError" exception.

Example:

bin(15)

bin(0xf)

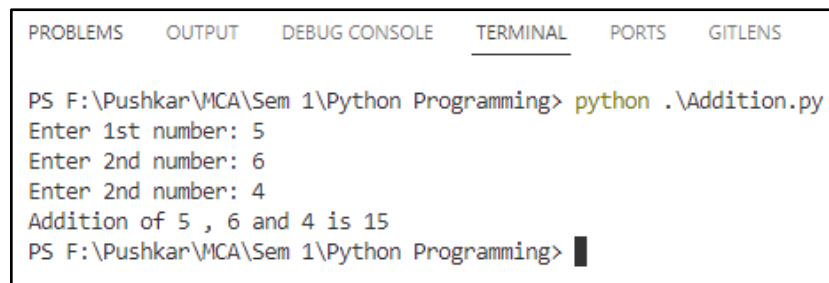
bin(0o17)

1) Add three numbers.**Code:**

```
number1 = int(input("Enter 1st number: "))
number2 = int(input("Enter 2nd number: "))
number3 = int(input("Enter 2nd number: "))
print("Addition of " , number1 , "," , number2 , "and" , number3 , "is" , (number1 + number2 + number3))
```

Conclusion:

The program for addition of three numbers was executed successfully.

Output:A screenshot of a terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, PORTS, and GITLENS. The TERMINAL tab is active, showing the command prompt PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\Addition.py. The program prompts for three numbers: Enter 1st number: 5, Enter 2nd number: 6, and Enter 2nd number: 4. It then outputs: Addition of 5 , 6 and 4 is 15. The prompt returns to PS F:\Pushkar\MCA\Sem 1\Python Programming>.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\Addition.py
Enter 1st number: 5
Enter 2nd number: 6
Enter 2nd number: 4
Addition of 5 , 6 and 4 is 15
PS F:\Pushkar\MCA\Sem 1\Python Programming> █
```

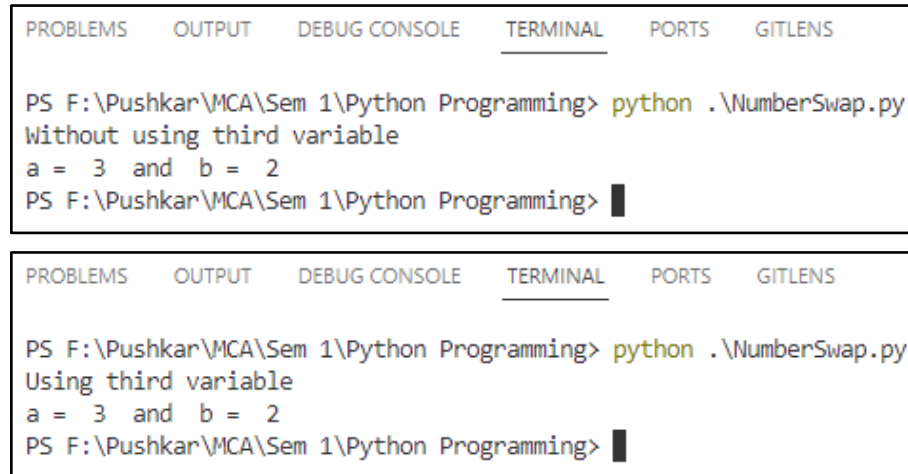
2) To swap two numbers using the third variable and without using the third variable.**Code:**

```
# Without using third variable
a = 2
b = 3
a, b = b, a
print("a = " , a, "b = " , b)

# Using third variable
a = 2
b = 3
temp = a
a = b
b = temp
print("a = " , a, " and " , "b = " , b)
```

Conclusion:

We swapped the values without using the third variable in the first program and using the third variable in the second program.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\NumberSwap.py
Without using third variable
a = 3 and b = 2
PS F:\Pushkar\MCA\Sem 1\Python Programming>

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

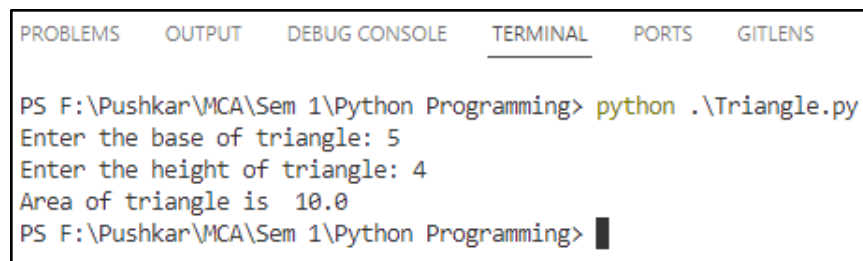
PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\NumberSwap.py
Using third variable
a = 3 and b = 2
PS F:\Pushkar\MCA\Sem 1\Python Programming>
```

3) Calculate area of triangle.**Code:**

```
x = 0.5
base = int(input("Enter the base of triangle: "))
height = int(input("Enter the height of triangle: "))
print("Area of triangle is ", (x * base * height))
```

Conclusion:

The area of the triangle is calculated on the input from the user.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

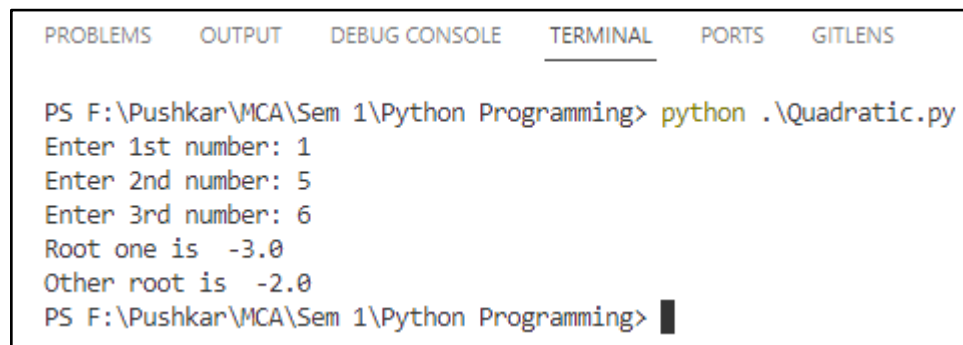
PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\Triangle.py
Enter the base of triangle: 5
Enter the height of triangle: 4
Area of triangle is 10.0
PS F:\Pushkar\MCA\Sem 1\Python Programming>
```

4) To solve Quadratic equations.**Code:**

```
a = int(input("Enter 1st number: "))
b = int(input("Enter 2nd number: "))
c = int(input("Enter 3rd number: "))
# Calculating determinant
x = (b ** 2) - (4 * a * c)
y = x ** 0.5
# Finding roots
if x < 0:
    print("The roots are imaginary.")
else:
    ans1 = (-b - y) / (2 * a)
    ans2 = (-b + y) / (2 * a)
    print("Root one is ", round(ans1, 2))
    print("Other root is ", round(ans2, 2))
```

Conclusion:

The roots of quadratic equations are calculated based on the user's input.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\Quadratic.py
Enter 1st number: 1
Enter 2nd number: 5
Enter 3rd number: 6
Root one is -3.0
Other root is -2.0
PS F:\Pushkar\MCA\Sem 1\Python Programming> █
```

5) To use Bitwise operators.**Code:**

```
a = int(input("Enter 1st number: "))
b = int(input("Enter 2nd number: "))
print("And Operator: ", a & b)
print("Or Operator: ", a | b)
print("Not Operator: ", ~a)
print("XOr Operator: ", (a ^ b))
print("Left shift", a << b)
print("Right shift", a >> b)
```

Conclusion:

We have demonstrated the use of various bitwise operators.

Output:

```
PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\bitwiseop.py
Enter 1st number: 2
Enter 2nd number: 3
And Operator: 2
Or Operator: 3
Not Operator: -3
XOr Operator: 1
Left shift 16
Right shift 0
PS F:\Pushkar\MCA\Sem 1\Python Programming> █
```

6) To compute compound interest given all the required values.**Code:**

```
# Taking values from user
principal = int(input("Enter principal amount: "))
interest = int(input("Enter interest rate: "))
time = int(input("Enter the duration: "))
# Calculating interest
amount = principal * pow(1 + interest / 100, time)
CI = amount - principal
print("The compound interest is ", CI)
```

Conclusion:

We have demonstrated the calculation of compound interest with the given values.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\cinterest.py
Enter principal amount: 100000
Enter interest rate: 7
Enter the duration: 2
The compound interest is 14490.0
PS F:\Pushkar\MCA\Sem 1\Python Programming> █
```

7) To generate a random number between 0 and 100.**Code:**

```
import random
print(random.randint(0, 100))
```

Conclusion:

We have demonstrated generating random number between 0 and 100 with the help of random module.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

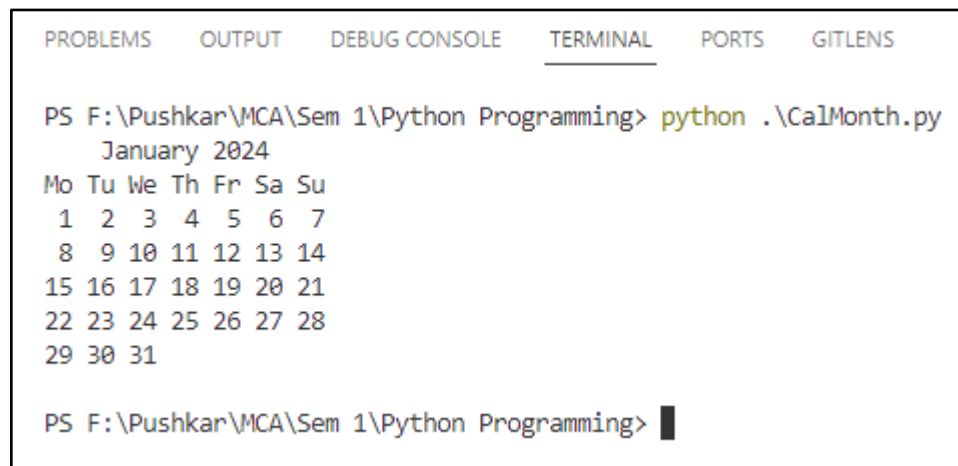
PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\RandNumber.py
60
PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\RandNumber.py
21
PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\RandNumber.py
84
PS F:\Pushkar\MCA\Sem 1\Python Programming> █
```

8) To display the calendar for January 2024.**Code:**

```
import calendar  
print(calendar.month(2024, 1))
```

Conclusion:

We have demonstrated displaying the current calendar year by using the calendar module.

Output:

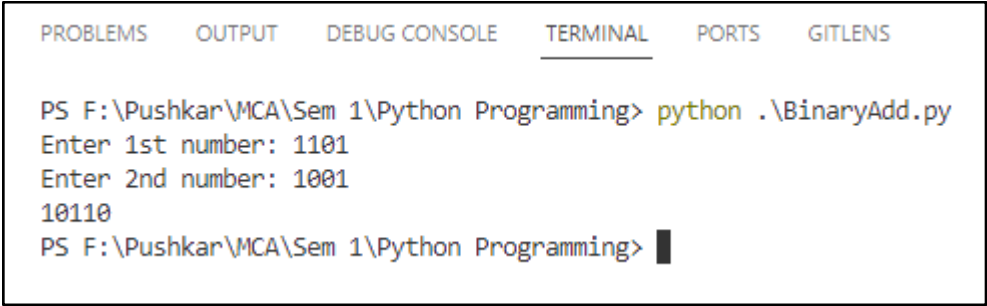
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  
  
PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\CalMonth.py  
January 2024  
Mo Tu We Th Fr Sa Su  
  1  2  3  4  5  6  7  
  8  9 10 11 12 13 14  
15 16 17 18 19 20 21  
22 23 24 25 26 27 28  
29 30 31  
  
PS F:\Pushkar\MCA\Sem 1\Python Programming> █
```

9) To add two binary numbers.**Code:**

```
a = input("Enter a binary number: ")  
b = input("Enter another binary number: ")  
print(bin(int(a, 2) + (int(b, 2))).replace("0b", " "))
```

Conclusion:

We have demonstrated the calculation of adding two binary numbers.

Output:

The screenshot shows a terminal window with a title bar containing tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is active), PORTS, and GITLENS. The terminal content shows a command prompt at 'PS F:\Pushkar\MCA\Sem 1\Python Programming>' followed by the command 'python .\BinaryAdd.py'. The program then prompts for two numbers: 'Enter 1st number: 1101' and 'Enter 2nd number: 1001'. The output of the program is '10110'. The prompt returns to 'PS F:\Pushkar\MCA\Sem 1\Python Programming>' with a cursor.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

PS F:\Pushkar\MCA\Sem 1\Python Programming> python .\BinaryAdd.py
Enter 1st number: 1101
Enter 2nd number: 1001
10110
PS F:\Pushkar\MCA\Sem 1\Python Programming> █
```

Name of Student: Pushkar Sane			
Roll Number: 45		Lab Assignment Number: 2	
Title of Lab Assignment: To implement Python programs with conditionals and loops.			
DOP:		DOS:	
CO Mapped: CO1	PO Mapped: PO3, PO5, PS01, PSO2	Faculty Signature:	Marks:

Practical No. 2

Aim: To implement Python programs with conditionals and loops.

Description:

- **Loops**

A loop in programming is a control structure that allows a set of instructions to be executed repeatedly based on a certain condition or for a specified number of times. Loops are essential for automating repetitive tasks, iterating over data structures, and controlling the flow of a program. They help streamline code and improve program efficiency by eliminating the need to write the same code multiple times. In addition to standard loops like for and while loops, Python also provides loop control statements like continue, break, and pass, which offer more fine-grained control over loop execution. These various loop constructs are fundamental for building flexible and efficient Python programs.

1. for Loop:

- The `for` loop is a fundamental looping construct used to iterate over a sequence of items. It is often used when you know how many times you want to repeat a block of code.
- You can use it to loop over sequences like lists, tuples, strings, dictionaries, and other iterable objects.
- The loop assigns each item from the sequence to a variable, and then the code block inside the loop is executed for each item.
- Example:

```
fruits = ["apple", "banana", "cherry"]
```

```
for fruit in fruits:
```

```
    print(fruit) # Output: apple banana cherry
```

2. while loop:

- The `while` loop is used for situations where you don't know in advance how many times the loop should run. It continues to execute as long as a specified condition is `True`.
- Be cautious with `while` loops to avoid infinite loops, where the condition is never `False`.

c. Example:

```
count = 0
while count < 5:
    print(count) #Output: 0 1 2 3 4
    count += 1
```

3. break Statement:

- a. The break statement is used to exit a loop prematurely, even if the loop condition is still True.
- b. It's often used when you want to terminate a loop based on a certain condition.

c. Example:

```
for i in range (10):
    if i == 5:
        break
    print(i) #Output: 0 1 2 3 4
```

4. Nested Loops:

- a. Python allows you to nest loops within each other, creating multi-level loops. This is useful when you need to iterate through multiple dimensions or combinations of items.
- b. Example:

```
for i in range (3):
    for j in range (2):
        print(f"({i}, {j})") #Output: (0, 0) (0, 1) (1, 0) (1, 1) (2, 0) (2, 1)
```

- **Loop Control Statements**

- 1. continue Statement:**

- a) The ``continue`` statement is used to skip the current iteration of a loop and move on to the next iteration.
- b) It's useful when you want to skip certain items or perform conditional skipping within a loop.

- c) Example:

```
for i in range(5):  
    if i == 2:  
        continue # Skip the current iteration when i is 2  
    print(i) # Output: 0 1 3 4
```

- 2. else Clause with Loops:**

- a) Python supports the ``else`` clause with ``for`` and ``while`` loops. The code in the ``else`` block is executed when the loop completes normally, i.e., when the loop condition becomes ``False`` or there are no more items to iterate.

- b) Example:

```
for i in range(5):  
    print(i) # Output: 0 1 2 3 4  
else:  
    print("Loop finished normally.") # Output: Loop finished normally.
```

- 3. pass Statement:**

- a) The ``pass`` statement is a no-op, which means it does nothing. It is often used as a placeholder when a statement is syntactically required, but you don't want any code to be executed.

- b) Example:

```
for i in range(3):  
    pass # No output, as the pass statement does nothing
```

These loops and loop control statements provide you with the flexibility to control the flow of your programs and perform repetitive tasks efficiently in Python.

- **Conditional Statements**

Conditional statements in Python allow you to control the flow of your program based on certain conditions. They enable you to make decisions and execute different blocks of code depending on whether a condition is true or false. Here are some of the main conditional statements in Python with examples:

1. if statements:

- a) The ``if`` statement is used to execute a block of code only if a specified condition is true.
- b) It can be followed by optional ``elif`` (else if) and ``else`` clauses for handling multiple conditions.

- c) Example:

```
x = 10
if x > 5:
    print ("x is greater than 5") # Output: x is greater than 5
elif x == 5:
    print ("x is equal to 5")
else:
    print ("x is less than 5")
```

2. if Expression (Ternary Operator):

- a) The ternary operator (``x if condition else y``) allows you to write a concise one-liner to assign a value to a variable based on a condition.

- b) Example:

```
age = 25
category = "Adult" if age >= 18 else "Minor"
print(category) # Output: Adult
```

3. while Loop with Condition:

- a) The while loop repeatedly executes a block of code as long as a specified condition is true.

- b) Example:

```
count = 0
while count < 5:
```

```
print(count) # Output: 0 1 2 3 4
count += 1
```

4. for Loop with Condition:

- a) The for loop can iterate over a sequence or iterable, and you can use the `if` statement within the loop to perform conditional actions.

- b) Example:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        print(f"{num} is even") # Output: 2 is even 4 is even
    else:
        print(f"{num} is odd") # Output: 1 is odd 3 is odd 5 is odd
```

5. assert Statement:

- a) The assert statement is used for debugging purposes to test if a given condition is `True`. If the condition is `False`, it raises an `AssertionError` exception.

- b) Example:

```
x = 10
assert x > 5, "x must be greater than 5" # No output if the assertion is true
```

6. in Operator:

- a) The `in` operator checks if a value exists in a sequence or collection. It is often used in `if` statements for membership testing.

- b) Example:

```
fruits = ["apple", "banana", "cherry"]
if "banana" in fruits:
    print("Banana is in the list") # Output: Banana is in the list
```

7. Nested Conditionals:

- a) You can nest conditional statements within each other to handle more complex decision-making scenarios.

- b) Example:

```
x = 10
if x > 5:
    if x % 2 == 0:
        print ("x is greater than 5 and even")
        # Output: x is greater than 5 and even
    else:
        print ("x is greater than 5 and odd")
```

Conditional statements are a fundamental part of programming, allowing you to create dynamic and responsive code that reacts to different situations. They are crucial for building logic and control in your Python programs

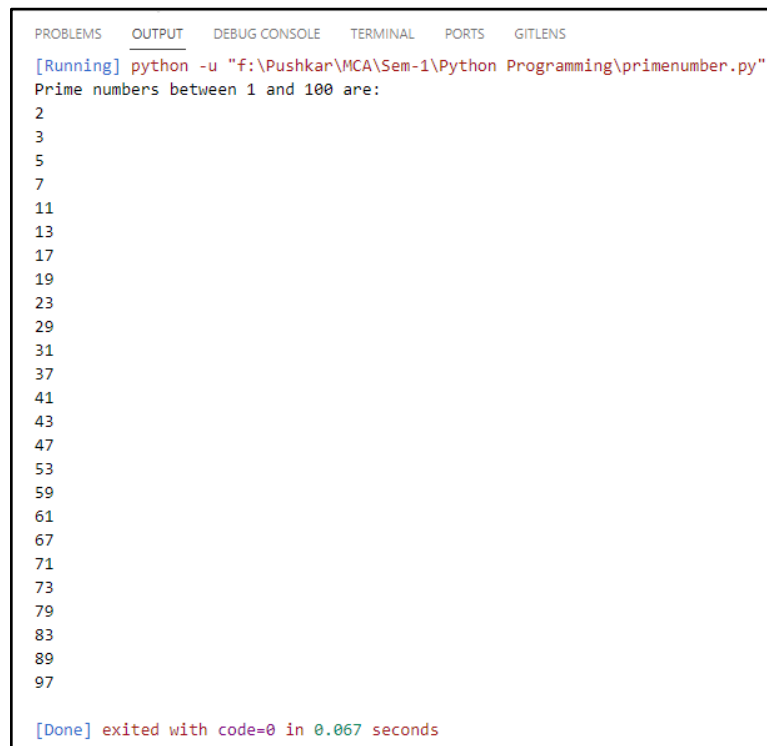
1) To find all the prime numbers in the interval 0 to 100.**Code:**

```
x = 1
y = 100

print("Prime numbers between", x, "and", y, "are:")
for num in range(x, y + 1):
    if num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                break
        else:
            print(num)
```

Conclusion:

Successfully demonstrated finding out prime numbers between 1 and 100

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\primenumber.py"
Prime numbers between 1 and 100 are:
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97

[Done] exited with code=0 in 0.067 seconds
```

2) To check if the given number is an Armstrong number or not.**Code:**

```
num = int(input("Enter a number: "))
sum = 0
temp = num

while temp > 0:
    digit = temp % 10
    sum += digit ** 3
    temp //= 10
if num == sum:
    print(num, "is an Armstrong number")
else:
    print(num, "is not an Armstrong number")
```

Conclusion:

Successfully executed the program for finding if the number is an armstrong number or not.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\armstrong.py
Enter a number: 417
It is not an Armstrong number
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\armstrong.py
Enter a number: 407
It is an Armstrong number
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

3) To check if the given char is a vowel or consonant.**Code:**

```
char = input("Enter an alphabet: ")
if char == 'A' or char == 'a' or char == 'I' or char == 'i' or char == 'O' or char == 'o' or
    char == 'U' or char == 'u':
    print(char + " is a vowel")
else:
    print(char + " is a consonant")
```

Conclusion:

Demonstrated the program for checking if the given character is a vowel or a consonant.

Output:

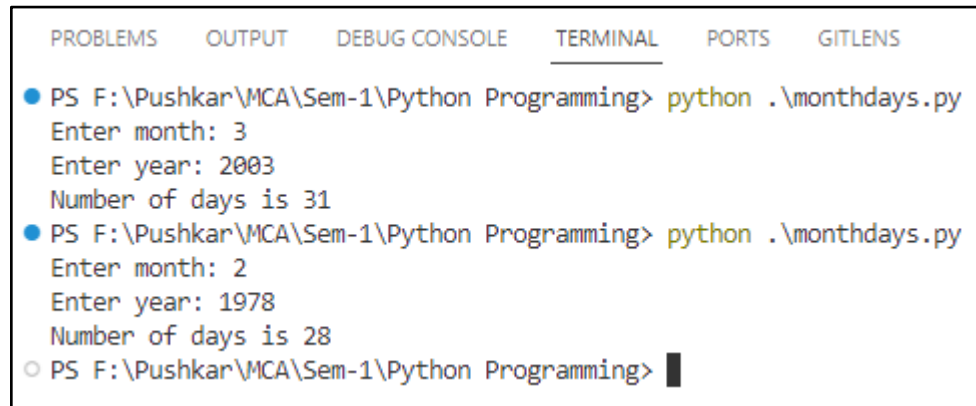
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\vowel.py
Enter an alphabet: A
A is a vowel
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\vowel.py
Enter an alphabet: a
a is a vowel
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\vowel.py
Enter an alphabet: x
x is a consonant
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

4) To convert a month to a number of days.**Code:**

```
import calendar
month = int(input("Enter month: "))
year = int(input("Enter year: "))
num_days = calendar.monthrange(year, month)[1]
print("Number of days is", num_days)
```

Conclusion:

Demonstrated the program for converting the month of a year to the number of days.

Output:

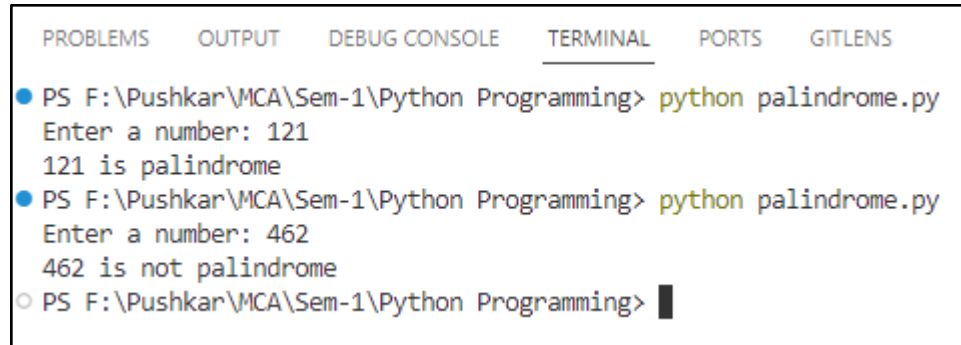
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\monthdays.py
Enter month: 3
Enter year: 2003
Number of days is 31
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\monthdays.py
Enter month: 2
Enter year: 1978
Number of days is 28
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

5) To check if a number is palindrome or not.**Code:**

```
num = int(input("Enter a number: "))
temp = num
reverse = 0
while temp > 0:
    remainder = temp % 10
    reverse = (reverse * 10) + remainder
    temp = temp // 10
if num == reverse:
    print(num, "is palindrome")
else:
    print(num, "is not palindrome")
```

Conclusion:

Demonstrated the program for checking if the given number is palindrome or not.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python palindrome.py
Enter a number: 121
121 is palindrome
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python palindrome.py
Enter a number: 462
462 is not palindrome
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

6) Write a program to take in the marks of 3 subjects and display the grade.

Code:

```
sub1 = int(input("Enter marks of English: "))
sub2 = int(input("Enter marks of Maths: "))
sub3 = int(input("Enter marks of Science: "))
```

```
avg = sub1 + sub2 + sub3 / 3
```

```
if avg >= 50:
```

```
    print("You got A grade.")
```

```
elif 40 <= avg < 50:
```

```
    print("You got B grade.")
```

```
elif 30 <= avg < 40:
```

```
    print("You got C grade.")
```

```
elif 20 <= avg < 20:
```

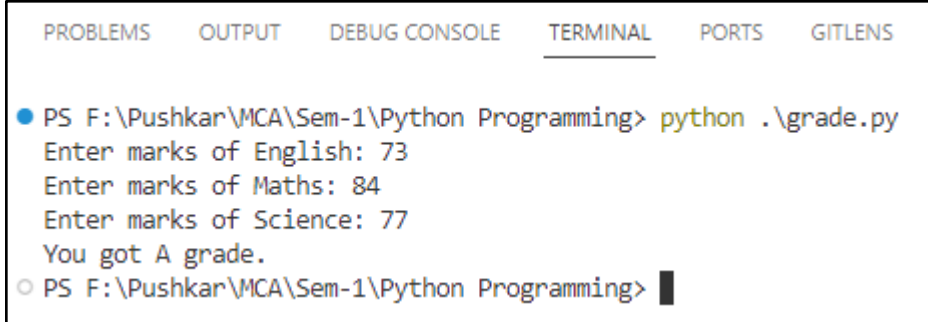
```
    print("You got D grade.")
```

```
else:
```

```
    print("You got failed.")
```

Conclusion:

Demonstrated the program for displaying grade on the basis of marks of 3 subjects.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\grade.py
Enter marks of English: 73
Enter marks of Maths: 84
Enter marks of Science: 77
You got A grade.
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

7) To add a matrix.**Code:**

```
X = [[12, 7, 3],
      [4, 5, 6],
      [7, 8, 9]]
```

```
Y = [[5, 8, 1],
      [6, 7, 3],
      [4, 5, 9]]
```

```
result = [[0, 0, 0],
           [0, 0, 0],
           [0, 0, 0]]
```

```
for i in range(len(X)):
    for j in range(len(X[0])):
        result[i][j] = X[i][j] + Y[i][j]
```

```
for r in result:
    print(r)
```

Conclusion:

Demonstrated the program for adding 2 matrices.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\matrix.py"
Matrix 1:  [[12, 7, 3], [4, 5, 6], [7, 8, 9]]
Matrix 2:  [[5, 8, 1], [6, 7, 3], [4, 5, 9]]

[17, 15, 4]
[10, 12, 9]
[11, 13, 18]

[Done] exited with code=0 in 0.05 seconds
```

8) To check the validity of password input by users, give 3 chances to the user to enter the correct password.

Validation:

- a) At least 1 letter between [a-z] and 1 letter between [A-Z].
- b) At least 1 number between [0-9].
- c) At least 1 character from [\$#@].
- d) Minimum length 6 characters.
- e) Maximum length 16 characters.

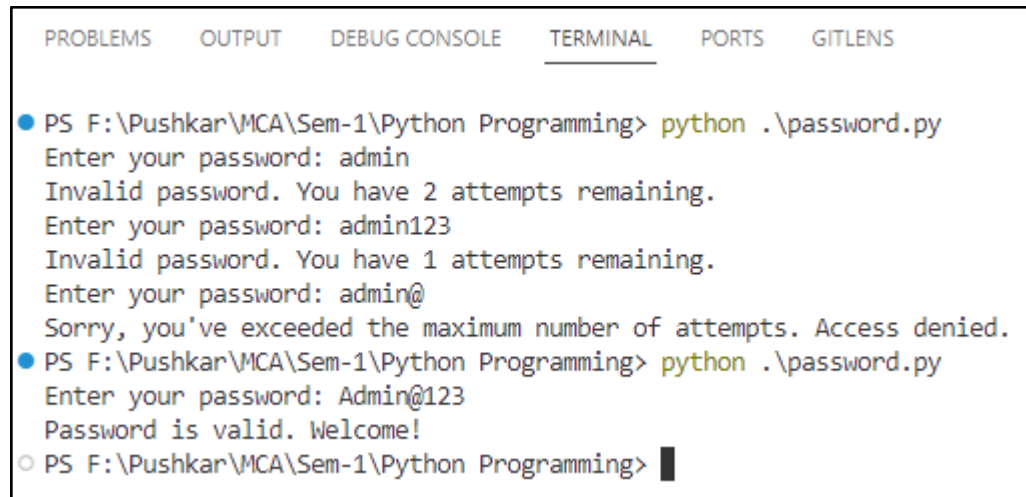
Code:

```
import re
max_attempts = 3
while max_attempts > 0:
    password = input("Enter your password: ")
    if 6 <= len(password) <= 16 and re.search("[a-z]", password) and re.search("[A-Z]", password) and re.search("[0-9]", password) and re.search("[$#@]", password):
        print("Password is valid. Welcome!")
        break
    else:
        max_attempts -= 1
        if max_attempts > 0:
            print(f"Invalid password. You have", max_attempts, "attempts remaining.")
        else:
```

```
print("Sorry, you've exceeded the maximum number of attempts. Access denied.")
```

Conclusion:

Demonstrated program of checking the validity of password given by user.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\password.py
Enter your password: admin
Invalid password. You have 2 attempts remaining.
Enter your password: admin123
Invalid password. You have 1 attempts remaining.
Enter your password: admin@
Sorry, you've exceeded the maximum number of attempts. Access denied.
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\password.py
Enter your password: Admin@123
Password is valid. Welcome!
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```


Name of Student: Pushkar Sane			
Roll Number: 45		Lab Assignment Number: 3	
Title of Lab Assignment: To implement Python programs using List, String, Set and Dictionary.			
DOP:		DOS:	
CO Mapped: CO1	PO Mapped: PO3, PO5, PS01, PSO2	Signature:	Marks:

Practical No. 3

Aim: To implement Python programs using List, String, Set and Dictionary.

Description:

In Python, `List`, `String`, `Set`, and `Dictionary` are all built-in data types that serve different purposes and have unique characteristics. Here's a description of each with examples:

1. List

- A list is an ordered collection of items, and it can contain elements of different data types.
- Lists are mutable, meaning you can change their contents after they are created.
- Lists are defined using square brackets `[]` and can contain zero or more elements separated by commas.
- Example:

```
my_list = [1, 2, 3, 4, 5]  
fruits = ["apple", "banana", "cherry"]  
mixed_list = [1, "hello", True, 3.14]
```

Some Basic List operations

a. Accessing Elements

Example:

```
first_element = my_list[0] # Access the first element (1)
```

```
last_element = my_list[-1] # Access the last element (5)
```

b. Slicing

Example:

```
sliced_list = my_list[1:4] # Creates a new list (2, 3, 4)
```

c. Appending and Extending

Example:

```
my_list.append(6) # Adds 6 to the end of the list
```

```
my_list.extend([7, 8]) # Extends the list with [7, 8]
```

d. Removing Elements

Example:

```
my_list.remove(3) # Removes the first occurrence of 3
```

```
popped_element = my_list.pop() # Removes and returns the last element
```

2. String

- A string is a sequence of characters, enclosed in either single (') or double (") quotes.
- Strings are immutable, which means once created, they cannot be changed.
- You can perform various string operations like concatenation, slicing, and formatting.

- Example:

```
my_string = "Hello, World!"
```

```
name = "Alice"
```

```
greeting = f"Hello, {name}!"
```

Some Basic String operations**a. String Concatenation**

Example:

```
greeting = "Hello, " + "Alice!"
```

b. String Length

Example:

```
length = len(my_string) # Returns the length of the string
```

c. Substring

Example:

```
substring = my_string[7:12] # Extracts "World"
```

d. String Methods

Example:

```
uppercase_string = my_string.upper() # Converts to uppercase
```

3. Set

- A set is an unordered collection of unique elements.
- Sets are defined using curly braces `{}` or the `set()` constructor.
- Sets are commonly used for tasks that involve testing membership or eliminating duplicates.
- Example:
`my_set = {1, 2, 3, 4, 5}`
`unique_characters = set("hello")`

Some Basic Set operations

a. Adding and Removing Elements

Example:

```
my_set.add(6) # Adds 6 to the set
```

```
my_set.remove(3) # Removes 3 from the set
```

b. Set Operations:

Example:

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
union_set = set1.union(set2) # Union of sets (1, 2, 3, 4, 5)
```

```
intersection_set = set1.intersection(set2) # Intersection (3)
```

4. Dictionary

- A dictionary is an unordered collection of key-value pairs.
- Each key in a dictionary is unique and maps to a specific value.
- Dictionaries are defined using curly braces `{}` with key-value pairs separated by colons `:`.
- Example:
`my_dict = {"name": "John", "age": 30, "city": "New York"}`
`student_scores = {"Alice": 95, "Bob": 87, "Charlie": 92}`

Some Basic Set operations**a. Accessing Values:**Example:

```
name = my_dict['name'] # Access the value associated with 'name'
```

b. Adding and Updating Key-Value Pairs:Example:

```
my_dict['occupation'] = 'Engineer' # Add a new key-value pair
```

```
my_dict['age'] = 31 # Update the value associated with 'age'
```

c. Removing Key-Value Pairs:Example:

```
del my_dict['city'] # Removes the 'city' key and its value
```

These data types are fundamental in Python and are used extensively in various programming tasks. Understanding when and how to use them is crucial for effective Python programming.

5. Bubble Sort Algorithm:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted. It's called "Bubble Sort" because the smaller (or larger, depending on the sorting order) elements "bubble" to the top of the list with each pass.

Here's a description of Bubble Sort in Python along with an example:

Bubble Sort Algorithm Steps:

- a. Start at the beginning of the list.
- b. Compare the first two elements. If the first element is greater (or smaller, depending on the sorting order) than the second, swap them.
- c. Move one position to the right.
- d. Repeat steps 2-3 until you reach the end of the list.
- e. Continue this process for each pair of adjacent elements, moving from the beginning to the end of the list.
- f. Repeat steps 1-5 until no more swaps are needed, indicating that the list is sorted.

1. To merge two lists and find the second largest element in the list using bubble sort.**Code:**

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

list1 = [1,3,8,9]
list2 = [7,4,7,6]

merged_list = list1 + list2
bubble_sort(merged_list)
second_largest = merged_list[-2]
print("Merged List:", merged_list)
print("Second Largest Element:", second_largest)
```

Conclusion:

The code successfully merges two lists (list1 and list2) into a single list (merged_list) and sorts it using the Bubble Sort algorithm. After sorting, it finds and prints the second largest element in the sorted merged list. Please note that Bubble Sort is not the most efficient sorting algorithm, especially for large lists, but it serves as an example for educational purposes.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\largest_element_bubble.py"
Merged List: [1, 3, 4, 6, 7, 7, 8, 9]
Second Largest Element: 8

[Done] exited with code=0 in 0.062 seconds
```

2. To calculate the no of uppercase, lowercase letters and digits in a string.**Code:**

```
input_string = "Hello World 12345"
upper_count=0
lower_count=0
digit_count=0
for char in input_string:
    if char.isupper():
        upper_count+=1
for char in input_string:
    if char.islower():
        lower_count+=1
for char in input_string:
    if char.isdigit():
        digit_count+=1
print("Uppercase letters:", upper_count)
print("Lowercase letters:", lower_count)
print("Digits:", digit_count)
```

Conclusion:

We have learned how to analyze a given input string in Python and count the occurrences of uppercase letters, lowercase letters, and digits using string methods and loops.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\calculate_letter.py"
Uppercase letters: 2
Lowercase letters: 8
Digits: 5

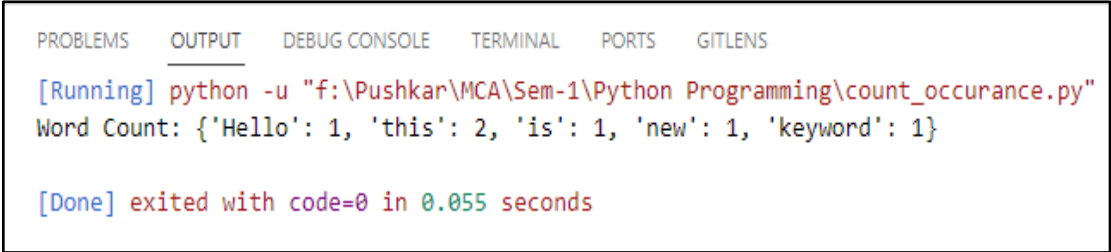
[Done] exited with code=0 in 0.058 seconds
```

3. To count the occurrences of each word in a given string sentence.**Code:**

```
sentence = "Hello this is new my keyword"
words = sentence.split()
word_count = {}
for word in words:
    word_count[word] = word_count.get(word, 0) + 1
print("Word Count:", word_count)
```

Conclusion:

In this code, we've learned how to tokenize a sentence into words, create a dictionary to count the frequency of each word, and print the word frequency count. This is a fundamental operation often used in natural language processing and text analysis tasks.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\count_occurance.py"
Word Count: {'Hello': 1, 'this': 2, 'is': 1, 'new': 1, 'keyword': 1}

[Done] exited with code=0 in 0.055 seconds
```


4. To add a key value pair to the dictionary and search and then delete the given key from the dictionary.**Code:**

```
my_dict = {}
my_dict['name'] = 'ABC'
my_dict['age'] = 31
my_dict['city'] = 'New York'
print("Dictionary after adding key-value pairs:")
print(my_dict)
search_key = 'age'
if search_key in my_dict:
    print(f"The value for key '{search_key}' is: {my_dict[search_key]}")
else:
    print(f"Key '{search_key}' not found in the dictionary")
delete_key = 'city'
if delete_key in my_dict:
    del my_dict[delete_key]
    print(f"Key '{delete_key}' deleted from the dictionary")
else:
    print(f"Key '{delete_key}' not found in the dictionary")
print("Dictionary after deleting a key:")
print(my_dict)
```

Conclusion:

In this code, we've learned essential dictionary operations, including adding, searching for, and deleting key-value pairs. This demonstrates how dictionaries are useful for organizing and manipulating data in Python.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\dict_op.py"
Dictionary after adding key-value pairs:
{'name': 'ABC', 'age': 31, 'city': 'New York'}
The value for key 'age' is: 31
Key 'city' deleted from the dictionary
Dictionary after deleting a key:
{'name': 'ABC', 'age': 31}

[Done] exited with code=0 in 0.059 seconds
```

5. Create one dictionary of 5 students with their name, address, age, class and marks of 5 subjects. Perform all the operations on the created dictionary.

Code:

Create a dictionary for 5 students

```
students = {
    'student1': {
        'name': 'ABC',
        'address': 'MU',
        'age': 18,
        'class': '11th',
        'marks': {
            'Mathematics': 85,
            'science': 92,
            'history': 78,
            'english': 88,
            'Drawing': 95
        }
    },

    'student2': {
        'name': 'DEF',
        'address': 'NA',
        'age': 17,
```

```
'class': '6th',  
'marks': {  
    'Mathematics': 90,  
    'science': 88,  
    'history': 76,  
    'english': 91,  
    'Drawing': 84  
}  
,
```

```
'student3': {  
    'name': 'GHI',  
    'address': 'CH',  
    'age': 19,  
    'class': '10th',  
    'marks': {  
        'Mathematics': 78,  
        'science': 85,  
        'history': 92,  
        'english': 80,  
        'Drawing': 89  
    }  
},
```

```
'student4': {  
    'name': 'JKL',  
    'address': 'MP',  
    'age': 17,  
    'class': '8th',  
    'marks': {  
        'Mathematics': 92,  
        'science': 84,  
        'history': 76,  
        'english': 90,
```

```
        'Drawing': 82
    }
},

'student5': {
    'name': 'MNO',
    'address': 'AG',
    'age': 18,
    'class': '9th',
    'marks': {
        'Mathematics': 88,
        'science': 90,
        'history': 85,
        'english': 87,
        'Drawing': 91
    }
}
}
```

```
print("Details of All the Students :")
# Display the information for each student
for student_id, student_info in students.items():
    print(f"Student ID: {student_id}")
    print(f"Name: {student_info['name']}")
    print(f"Address: {student_info['address']}")
    print(f"Age: {student_info['age']}")
    print(f"Class: {student_info['class']}")
    print("Marks:")

    for subject, marks in student_info['marks'].items():
        print(f"{subject}: {marks}")
    print()
```

```
# Search for a student by ID
search_id = 'student3'
if search_id in students:
    print(f"Student ID: {search_id}")
    student_info = students[search_id]
    print(f"Name: {student_info['name']}")
else:
    print(f"Student with ID '{search_id}' not found")

# Delete a student by ID
delete_id = 'student4'
if delete_id in students:
    del students[delete_id]
    print(f"Student with ID '{delete_id}' deleted from the dictionary")
else:
    print(f"Student with ID '{delete_id}' not found in the dictionary")
```

Conclusion:

In this code, we've learned about dictionaries, nested dictionaries, and how to manipulate and access data within them. It's a practical example of organizing and managing structured data in Python, which is essential for various data processing and management tasks.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\stud_dict.py"
Details of All the Students :
Student ID: student1
Name: ABC
Address: MU
Age: 18
Class: 11th
Marks:
Student ID: student2
Name: DEF
Address: NA
Age: 17
Class: 6th
Marks:
Student ID: student3
Name: GHI
Address: CH
Age: 19
Class: 10th
Marks:
Student ID: student4
Name: JKL
Address: MP
Age: 17
Class: 8th
Marks:
Student ID: student5
Name: MNO
Address: AG
Age: 18
Class: 9th
Marks:
Mathematics: 88

science: 90

history: 85

english: 87

Drawing: 91

Student ID: student3
Name: GHI
Student with ID 'student4' deleted from the dictionary

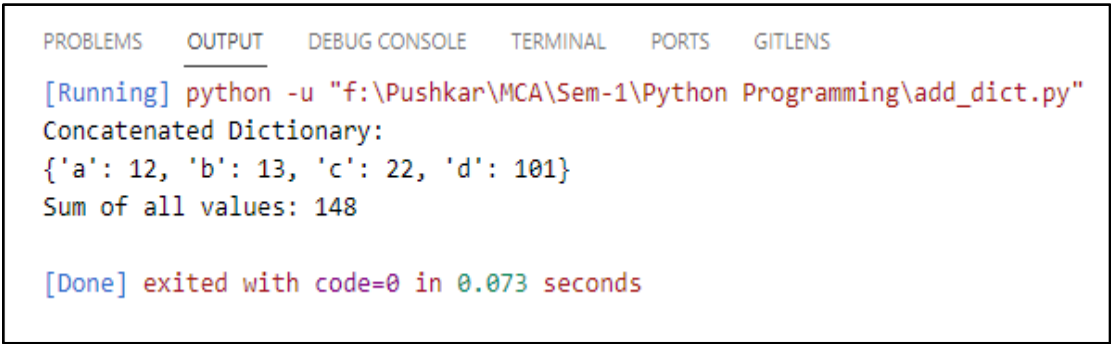
[Done] exited with code=0 in 0.073 seconds
```

6. To concatenate two dictionaries and find the sum of all values in the dictionary.**Code:**

```
# Define two dictionaries
dict1 = {'a': 12, 'b': 59, 'c': 71}
dict2 = {'b': 13, 'c': 22, 'd': 101}
# Concatenate the two dictionaries
concatenated_dict = {**dict1, **dict2}
# Calculate the sum of all values in the concatenated dictionary
total_sum = sum(concatenated_dict.values())
# Display the concatenated dictionary and the sum
print("Concatenated Dictionary:")
print(concatenated_dict)
print(f"Sum of all values: {total_sum}")
```

Conclusion:

In this code, we've learned how to merge two dictionaries into a single concatenated dictionary and calculate the sum of values within it. This is a useful technique for combining data from multiple sources and performing aggregate operations on the merged data

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\add_dict.py"
Concatenated Dictionary:
{'a': 12, 'b': 13, 'c': 22, 'd': 101}
Sum of all values: 148

[Done] exited with code=0 in 0.073 seconds
```

7. To add and remove elements from set and perform all the set operations like Union, Intersection, Difference and Symmetric Difference.**Code:**

```
# Create two sets
set1 = {2, 3, 4, 5, 6}
set2 = {4, 5, 6, 7, 8}
# Add an element to a set
set1.add(6)
print("After adding 6 to set1:", set1)
# Remove an element from a set
set2.remove(7)
print("After removing 7 from set2:", set2)
# Union of two sets
union_result = set1.union(set2)
print("Union of set1 and set2:", union_result)
# Intersection of two sets
intersection_result = set1.intersection(set2)
print("Intersection of set1 and set2:", intersection_result)
# Difference of two sets
difference_result = set1.difference(set2)
print("Difference of set1 and set2:", difference_result)
# Symmetric Difference of two sets
symmetric_difference_result = set1.symmetric_difference(set2)
print("Symmetric Difference of set1 and set2:", symmetric_difference_result)
```

Conclusion:

In this code, we've learned how to perform common set operations such as adding, removing, finding the union, intersection, difference, and symmetric difference of sets. Sets are useful for dealing with unique and unordered collections of elements in Python.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\set_op.py"
After adding 6 to set1: {2, 3, 4, 5, 6}
After removing 7 from set2: {4, 5, 6, 8}
Union of set1 and set2: {2, 3, 4, 5, 6, 8}
Intersection of set1 and set2: {4, 5, 6}
Difference of set1 and set2: {2, 3}
Symmetric Difference of set1 and set2: {2, 3, 8}

[Done] exited with code=0 in 0.058 seconds
```

8. Perform different operations on Tuple.**Code:**

```
# Creating a tuple
my_tuple = (10, 11, 12, 13, 14)

# Accessing elements
print("Accessing elements:")
print(my_tuple[0]) # Access the first element (1)
print(my_tuple[-1]) # Access the last element (5)

# Slicing
print("\nSlicing:")
sliced_tuple = my_tuple[1:4] # Creates a new tuple (2, 3, 4)
print(sliced_tuple)

# Concatenating tuples
print("\nConcatenating tuples:")
tuple1 = (1, 2)
tuple2 = (3, 4)
concatenated_tuple = tuple1 + tuple2 # Creates a new tuple (1, 2, 3, 4)
print(concatenated_tuple)

# Tuple repetition
print("\nTuple repetition:")
repeated_tuple = my_tuple * 2 # Creates a new tuple (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
print(repeated_tuple)
```

```
# Finding tuple length
print("\nFinding tuple length:")
length = len(my_tuple) # Returns 5
print(length)

# Iterating through a tuple
print("\nIterating through a tuple:")
for item in my_tuple:
    print(item)

# Checking membership
print("\nChecking membership:")
if 12 in my_tuple:
    print("3 is in the tuple")

# Tuple unpacking
print("\nTuple unpacking:")
a, b, c, d, e = my_tuple
print(f"a: {a}, b: {b}, c: {c}, d: {d}, e: {e}")

# Count and index
print("\nCount and index:")
count = my_tuple.count(3) # Returns the count of 3 in the tuple
index = my_tuple.index(12) # Returns the index of the first occurrence of 4
print(f"Count of 3: {count}")
print(f"Index of 4: {index}")
```

Conclusion:

In this code, we've learned various operations and techniques for working with tuples in Python. Tuples are immutable, ordered collections of elements and can be useful in scenarios where data should not be modified after creation.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\tuple_op.py"
Accessing elements:
10
14

Slicing:
(11, 12, 13)

Concatenating tuples:
(1, 2, 3, 4)

Tuple repetition:
(10, 11, 12, 13, 14, 10, 11, 12, 13, 14)

Finding tuple length:
5

Iterating through a tuple:
10
11
12
13
14

Checking membership:
3 is in the tuple

Tuple unpacking:
a: 10, b: 11, c: 12, d: 13, e: 14

Count and index:
Count of 3: 0
Index of 4: 2

[Done] exited with code=0 in 0.063 seconds
```

9. Write a Python program to count the elements in a list until an element is a tuple.**Code:**

```
my_list = [11, 22, 33, 'world', (43, 54), 62, 71]
```

```
# Initialize a counter
```

```
count = 0
```

```
# Iterate through the list
```

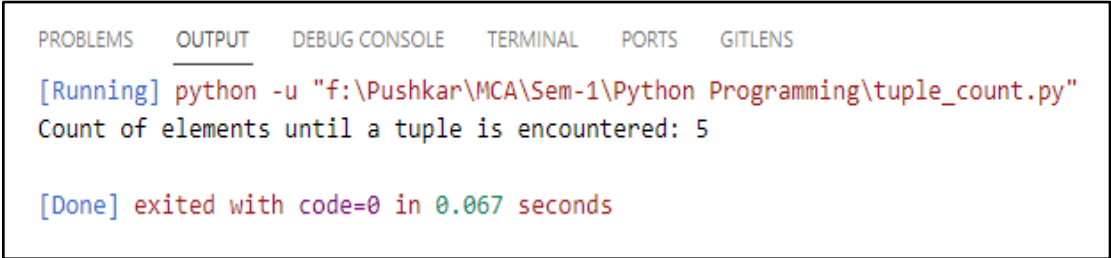
```
for item in my_list:
```

```
count += 1
if isinstance(item, tuple):
    break
```

```
# Print the count of elements until a tuple is encountered
print(f"Count of elements until a tuple is encountered: {count}")
```

Conclusion:

In this code, we've learned how to iterate through a list and count the number of elements until a specific condition is met (in this case, until a tuple is encountered). This demonstrates how to use a for loop, conditionals, and the break statement for control flow in Python.

Output:

The screenshot shows a code editor with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, PORTS, and GITLENS. The OUTPUT tab is active, displaying the following text:

```
[Running] python -u "f:\Pushkar\MCA\Sem-1\Python Programming\tuple_count.py"
Count of elements until a tuple is encountered: 5

[Done] exited with code=0 in 0.067 seconds
```

Name of Student: Pushkar Sane			
Roll Number: 45		Lab Assignment Number: 4	
Title of Lab Assignment: To implement programs on Python Functions and Modules.			
DOP:		DOS:	
CO Mapped: CO1	PO Mapped: PO3, PO5, PS01, PS02	Signature:	Marks:

Practical No. 4

Aim: To implement programs on Python Functions and Modules.

Description:

- **Function In Python:**

- a. A function in Python is a named block of code that performs a specific task or set of tasks.
- b. Functions are essential for organizing and structuring your code, promoting code reuse, and making your code more readable and maintainable. They encapsulate a series of instructions, allowing you to call and reuse them throughout your program.

- **Function Syntax:**

Here's the general syntax of defining a function in Python:

```
def function_name(parameter1, parameter2, ...):  
    # Function body - code to perform the task  
    # You can use parameters in the function body  
    result = parameter1 + parameter2  
    return result # Optional, specifies the value to be returned
```

- a. **def:** This keyword is used to define a function in Python.
- b. **function_name:** Choose a descriptive name for your function that reflects its purpose. Function names should follow naming conventions (e.g., use lowercase letters and underscores).
- c. **parameter1, parameter2, ... :** These are optional input parameters or arguments that the function accepts. Parameters allow you to pass data into the function for processing.
- d. **':' :** A colon is used to denote the beginning of the function's body.
- e. **Function body:** The function body consists of the actual code that performs the desired task. It can include calculations, loops, conditionals, and other statements.
- f. **return:** This keyword, followed by an expression, is used to specify the value that the function should return as its result. Not all functions need to return a value; it's optional.

Example: Adding Two Numbers:

```
def add_numbers(num1, num2):  
    result = num1 + num2  
    return result  
  
# Call the function with arguments 5 and 3  
sum_result = add_numbers(5, 3)  
print(sum_result) # Output: 8  
  
# Call the function with different arguments  
result2 = add_numbers(10, 20)  
print(result2) # Output: 30
```

- **Recursion:**

Recursion is a programming technique in which a function calls itself to solve a problem. In Python, you can implement recursive functions to break down complex problems into simpler, self-similar sub problems. To successfully use recursion, you need to define a base case (the terminating condition) and one or more recursive cases (the cases where the function calls itself).

Here's a detailed explanation of recursion with an example:

Factorial Calculation Using Recursion:

```
def factorial(n):  
    # Base case: When n is 0 or 1, the factorial is 1.  
    if n == 0 or n == 1:  
        return 1  
    # Recursive case: Calculate factorial by calling the function recursively.  
    else:  
        return n * factorial(n - 1)
```

In this recursive function:

- a. The base case is when `n` is either `0` or `1`. In these cases, we return `1` because `0!` and `1!` are both equal to `1`.

- b. The recursive case calculates the factorial of `n` by calling the `factorial` function recursively with the argument `n - 1`. This reduces the problem to a smaller subproblem.

- **Lambda Function**

In Python, a lambda function is a small, anonymous, and inline function defined using the lambda keyword. Lambda functions are also known as anonymous functions because they don't have a name. They are typically used for short, simple operations where defining a full function using the def keyword would be overly verbose.

The basic syntax of a lambda function is as follows:

lambda arguments: expression

- a. **lambda:** The lambda keyword is used to indicate the creation of a lambda function.
- b. **arguments:** These are the input parameters to the lambda function, separated by commas. Lambda functions can take any number of arguments but are often used with one or two.
- c. **expression:** This is a single expression that the lambda function evaluates and returns as its result.

Here's an example to illustrate how lambda functions work:

```
# Regular function to add two numbers
```

```
def add(x, y):
```

```
    return x + y
```

```
# Equivalent lambda function
```

```
add_lambda = lambda x, y: x + y
```

```
# Using both functions to add numbers
```

```
result1 = add(5, 3)
```

```
result2 = add_lambda(5, 3)
```

```
print("Using the regular function:", result1) # Output: Using the regular function: 8
```

```
print("Using the lambda function:", result2)
```

- **Module:**

In Python, a module is a file containing Python code, including variables, functions, and classes, that can be imported and used in other Python scripts or programs. Modules allow you to organize your code into reusable and manageable units, making your code more

modular and maintainable. A file in Python can serve as a module if it contains Python code. To create a module, you typically save your Python code in a `.py` file with the same name as the module you want to create. For example, if you want to create a module named `my_module`, you could create a file named `my_module.py`.

Here's a simple example of a Python module:

my_module.py

```
# This is a Python module named 'my_module'
```

```
def greet(name):
```

```
    return f"Hello, {name}!"
```

```
def add(a, b):
```

```
    return a + b
```

```
pi = 3.14159265359
```

Now, you can use this module in another Python script by importing it using the `import` statement:

main.py

```
# Importing the 'my_module' module
```

```
import my_module
```

```
# Using functions and variables from the module
```

```
print(my_module.greet("Alice"))    # Output: Hello, Alice!
```

```
print(my_module.add(5, 3))         # Output: 8
```

```
print(my_module.pi)               # Output: 3.14159265359
```

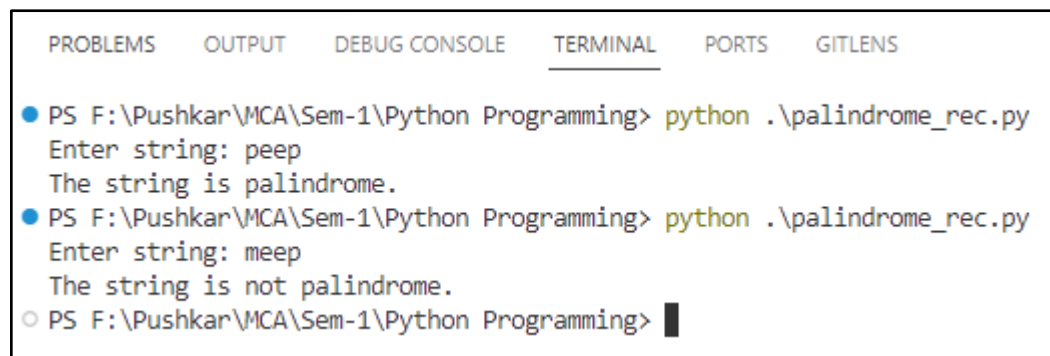
1. To check whether a string is palindrome or not using function recursion.**Code:**

```
# Define a function
def isPalindrome(string):
    if len(string) < 1:
        return True
    else:
        if string[0] == string[-1]:
            return isPalindrome(string[1:-1])
        else:
            return False
#Enter input string
str1 = input("Enter string:")

if(isPalindrome(str1)==True):
    print("The string is palindrome.")
else:
    print("The string is not palindrome.")
```

Conclusion:

Here, we reverse a string using the recursion method which is a process where the function calls itself. Then we check if the reversed string matches with the original string and demonstrate the program for checking if a string is palindrome or not using recursion.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\palindrome_rec.py
Enter string: peep
The string is palindrome.
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\palindrome_rec.py
Enter string: meep
The string is not palindrome.
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

2. To find Fibonacci series using recursion.

Code:

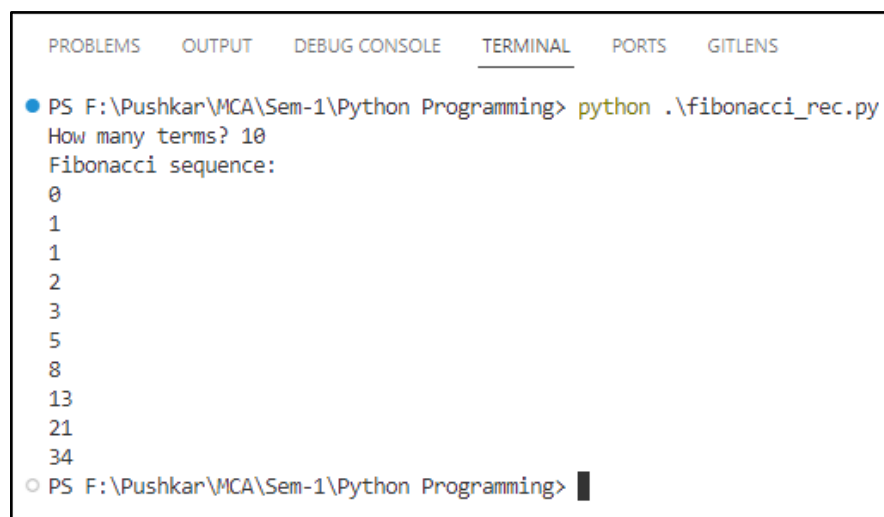
```
def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))

# take input from the user
nterms = int(input("How many terms? "))
# check if the number of terms is valid
if nterms <= 0:
    print("Please enter a positive integer")
else:
    print("Fibonacci sequence:")
    for i in range(nterms):
        print(recur_fibo(i))
```

Conclusion:

In this program, we store the number of terms to be displayed in nterms. A recursive function recur_fibo() is used to calculate the nth term of the sequence. We use a for loop to iterate and calculate each term recursively.

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\fibonacci_rec.py
How many terms? 10
Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

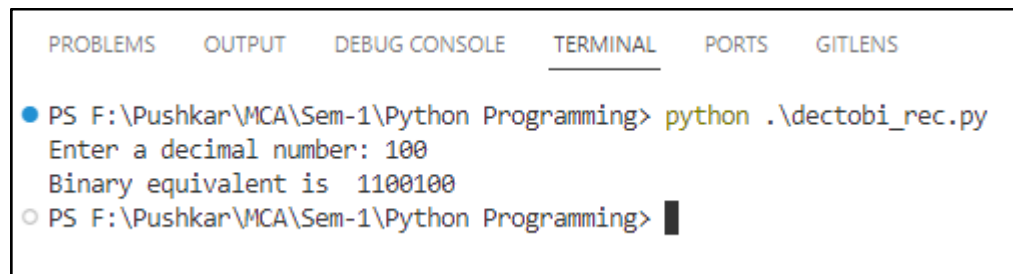
3. To find the binary equivalent of a number using recursion.**Code:**

```
# Decimal to binary conversion using recursion
def find( decimal_number ):
    if decimal_number == 0:
        return 0
    else:
        return (decimal_number % 2 + 10 *
                find(int(decimal_number // 2)))

# Driver Code
decimal_number = int(input("Enter a decimal number: "))
print(find(decimal_number))
```

Conclusion:

Demonstrated the program for finding the binary equivalent or converting a decimal number to binary number using recursion.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\dectobi_rec.py
Enter a decimal number: 100
Binary equivalent is 1100100
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

4. To use lambda function on list to generate filtered list, mapped list and reduced list.**Code:**

```
from functools import reduce

# Sample list
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filter using a lambda function
filtered_list = list(filter(lambda x: x % 2 == 0, my_list))

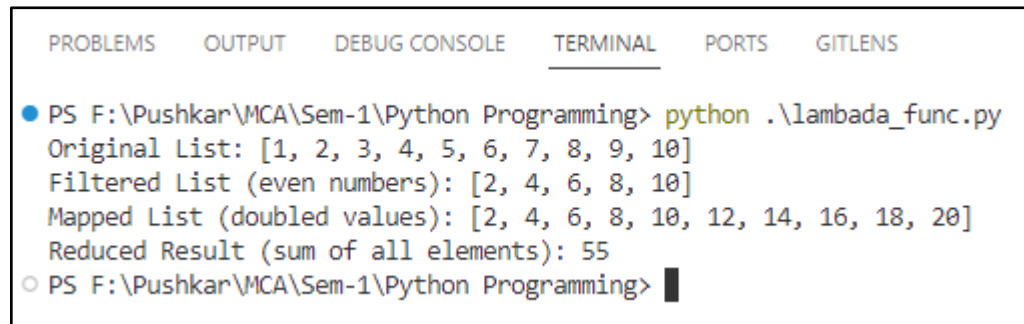
# Map using a lambda function
mapped_list = list(map(lambda x: x * 2, my_list))

# Reduce using a lambda function
reduced_result = reduce(lambda x, y: x + y, my_list)

# Print the results
print("Original List:", my_list)
print("Filtered List (even numbers):", filtered_list)
print("Mapped List (doubled values):", mapped_list)
print("Reduced Result (sum of all elements):", reduced_result)
```

Conclusion

In this Python program, we demonstrated how to use lambda functions in conjunction with the filter(), map(), and reduce() functions to perform different operations on a given list.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

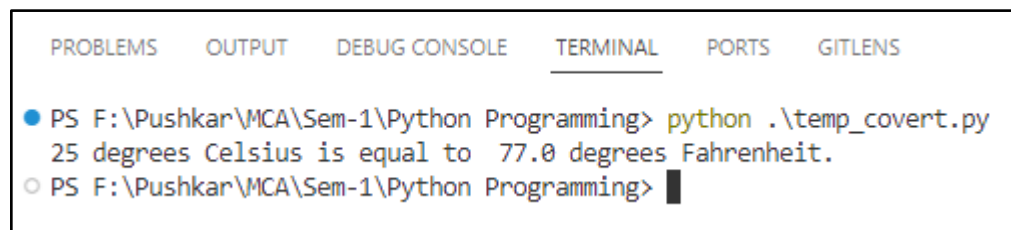
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\lambda_func.py
Original List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Filtered List (even numbers): [2, 4, 6, 8, 10]
Mapped List (doubled values): [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Reduced Result (sum of all elements): 55
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

5. Convert the temperature in Celsius to Fahrenheit in a list using an anonymous function.**Code:**

```
# Convert Celsius to Fahrenheit using a lambda function
celsius_to_fahrenheit = lambda c: (c * 9/5) + 32
# Temperature in Celsius
celsius_temperature = 25
# Use the lambda function to convert
fahrenheit_temperature = celsius_to_fahrenheit(celsius_temperature)
# Print the result
print(celsius_temperature, "degrees Celsius is equal to ", fahrenheit_temperature ,
      "degrees Fahrenheit.")
```

Conclusion:

This program demonstrates how to perform a straightforward temperature conversion on a list using an anonymous lambda function and then display the results. It's a concise way to achieve the desired conversion without the need for additional functions or complex string formatting.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\temp_convert.py
  25 degrees Celsius is equal to 77.0 degrees Fahrenheit.
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

6. To create modules in python and access functions of the module by importing it to another file/module. (Calculator program)

Code:

Calculator.py

```
# calculator.py
def add(x, y):
    return x + y
def subtract(x, y):
    return x - y
def multiply(x, y):
    return x * y
def divide(x, y):
    if y == 0:
        return "Error: Division by zero"
    return x / y
```

Main.py

```
# main.py
import calculator
def main():
    print("Simple Calculator")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    choice = input("Enter operation (1/2/3/4): ")
    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))
    if choice == '1':
        result = calculator.add(num1, num2)
        print(f"Result: {result}")
    elif choice == '2':
        result = calculator.subtract(num1, num2)
        print(f"Result: {result}")
```

```
elif choice == '3':
    result = calculator.multiply(num1, num2)
    print(f"Result: {result}")
elif choice == '4':
    result = calculator.divide(num1, num2)
    print(f"Result: {result}")
else:
    print("Invalid choice")
if __name__ == "__main__":
    main()
```

Conclusion:

Here, we've demonstrated the creation of a modular calculator program using two separate files. The calculator.py module encapsulates core arithmetic functions, while the main.py script serves as a user interface, importing and utilizing these functions. This modular approach enhances code organization, reusability, and readability, making it easier to maintain and expand the calculator program.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> cd .\calculator\
● PS F:\Pushkar\MCA\Sem-1\Python Programming\calculator> python .\main.py
Simple Calculator
1. Add
2. Subtract
3. Multiply
4. Divide
Enter operation (1/2/3/4): 1
Enter first number: 2
Enter second number: 3
Result: 5.0
○ PS F:\Pushkar\MCA\Sem-1\Python Programming\calculator> █
```


Name of Student: Pushkar Sane			
Roll Number: 45		Lab Assignment Number: 5	
Title of Lab Assignment: To implement programs on OOP Concepts in python.			
DOP:		DOS:	
CO Mapped: CO2	PO Mapped: PO5, PSO1	Signature:	Marks:

Practical No. 5

Aim: Implement programs on OOP Concepts in python.

1. Python Program to Create a Class and Compute the Area and the Perimeter of the Circle.
2. To Implement Multiple Inheritance in python.
3. To Implement a program with the same method name and multiple arguments.
4. To Implement Operator Overloading in python.
5. Write a program which handles various exceptions in python.

Description:

Object-Oriented Programming (OOP) is a programming paradigm that focuses on organizing code into objects, which are instances of classes. Python is an object-oriented programming language, and it supports the fundamental OOP concepts.

Here are the key OOP concepts in Python:

1. Classes and Objects:

- A class is a blueprint or template for creating objects.
- An object is an instance of a class.
- Classes define attributes (data) and methods (functions) that the objects created from them will have.
- Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
# Creating objects from the class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
# Accessing object attributes and methods
print(person1.name) # Alice
person2.say_hello() # Hello, my name is Bob and I am 25 years old.
```

2. Inheritance:

- Inheritance allows you to create a new class that is based on an existing class. The new class inherits the attributes and methods of the base class.
- Python supports single and multiple inheritance.
- Example:

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id
    def study(self):
        print(f"{self.name} is studying.")
student = Student("Charlie", 20, "12345")
student.say_hello()    # Inherits from the Person class
student.study()        # Additional method specific to Student class
```

3. Encapsulation:

- Encapsulation is the concept of restricting access to certain parts of an object or class, typically by using private and public access modifiers.
- In Python, there are no true private variables or methods, but you can use naming conventions (e.g., prefixing with an underscore) to indicate privacy.
- Example:

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance # Private attribute
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
    def withdraw(self, amount):
        if amount > 0 and amount <= self._balance:
            self._balance -= amount
    def get_balance(self):
        return self._balance    # Public method to access balance
account = BankAccount(1000)
account.deposit(500)
```

```
account.withdraw(200)
print(account.get_balance()) # 1300
```

4. Polymorphism

- Polymorphism allows objects of different classes to be treated as objects of a common base class.
- It simplifies code by allowing you to work with objects generically without needing to know their specific types.
- Example:

```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length * self.side_length

shapes = [Circle(5), Square(4)]
for shape in shapes:
    print(f"Area: {shape.area()}")
```

These are the core OOP concepts in Python. They help you create modular, reusable, and organized code by promoting the use of objects and classes.

5. Overloading:

In Python, method overloading is not supported in the same way it is in some other programming languages, like Java or C++. Method overloading typically refers to defining multiple methods with the same name in a class but with different parameter lists. In Python, if you define multiple methods with the same name in a class, the last one defined

will override any previous ones, and you can't call the overloaded method with different argument types as you can in languages that support method overloading.

Example:

```
class MyClass:
    def my_method(self, param1, param2=None):
        if param2 is not None:
            # Do something when both parameters are provided
            result = param1 + param2
        else:
            # Do something when only one parameter is provided
            result = param1
        return result
obj = MyClass()
result1 = obj.my_method(5)
result2 = obj.my_method(5, 10)
print(result1) # Output: 5
print(result2) # Output: 15
```

In the example above, the `my_method` function is defined with two parameters, but the second parameter has a default value of `None`. This allows you to call the method with one or two arguments, and the method behaves differently depending on the number of arguments provided.

Keep in mind that this approach is more Pythonic and flexible than traditional method overloading found in some other languages because it allows you to adapt the method's behavior at runtime. It's also easier to read and understand.

In Python, exceptions are used to handle errors and unexpected situations in your code. Here are some common exceptions in Python and examples of how they can occur:

6. Exceptions in Python:

- a. **Syntax Error:** Occurs when there is a syntax error in your code.

Example:

```
def some_function()
```

```
# Missing colon at the end of the function definition
print("Hello, World!")
```

- b. IndentationError:** Occurs when there is an issue with the indentation of your code.

Example:

```
if True:
print("This line is not properly indented.")
```

- c. NameError:** Occurs when a local or global name is not found.

Example:

```
print(variable_that_does_not_exist)
```

- d. TypeError:** Occurs when you try to perform an operation on incompatible data types.

Example:

```
num = "5"
result = num + 7 # Trying to add a string and an integer
```

- e. ValueError:** Occurs when a function receives an argument of the correct data type but an inappropriate value.

Example:

```
int("abc") # Converting a non-numeric string to an integer
```

- f. IndexError:** Occurs when you try to access an index that is out of range in a sequence (e.g., a list or string).

Example:

```
my_list = [1, 2, 3]
print(my_list[5]) # Accessing an index that doesn't exist
```

- g. KeyError:** Occurs when you try to access a dictionary key that doesn't exist.

Example:

```
my_dict = {"name": "John", "age": 30}
print(my_dict["city"]) # Accessing a key that is not in the dictionary
```

- h. FileNotFoundError:** Occurs when you try to open a file that does not exist.

Example:

```
with open("nonexistent_file.txt", "r") as file:
```

```
    data = file.read()
```

- i. ZeroDivisionError:** Occurs when you attempt to divide by zero.

Example:

```
result = 5 / 0
```

- j. ImportError:** Occurs when there is an issue with importing a module or library.

Example:

```
import non_existent_module
```

- k. AttributeError:** Occurs when you try to access an attribute or method that does not exist.

Example:

```
class MyClass:
```

```
    def __init__(self, value):
```

```
        self.value = value
```

```
obj = MyClass(42)
```

```
print(obj.non_existent_attribute)
```

- l. Custom Exceptions:** You can also define and raise your own custom exceptions.

Example:

```
class MyCustomException(Exception):
```

```
    def __init__(self, message):
```

```
        super().__init__(message)
```

```
        raise MyCustomException("This is a custom exception.")
```

Handling exceptions allows you to gracefully manage errors in your code, preventing it from crashing and providing meaningful feedback to users or developers. You can use ``try``, ``except``, ``finally``, and other control structures to manage exception handling in Python.

1. Python Program to Create a Class and Compute the Area and the Perimeter of the Circle.

Code:

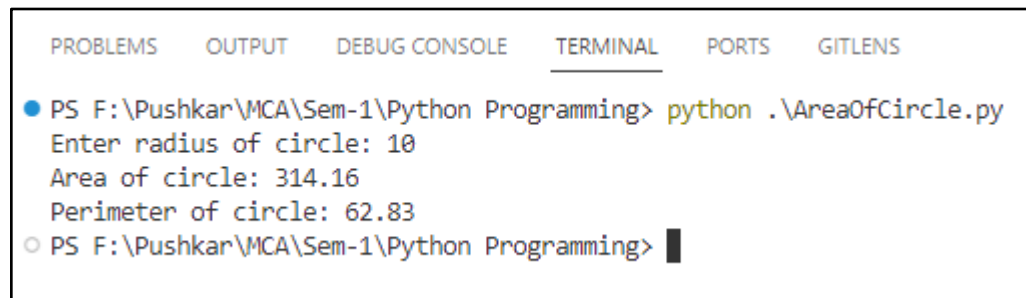
```
import math
class circle():
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * (self.radius ** 2)
    def perimeter(self):
        return 2 * math.pi * self.radius

r = int(input("Enter radius of circle: "))
obj = circle(r)
print("Area of circle:", round(obj.area(), 2))
print("Perimeter of circle:", round(obj.perimeter(), 2))
```

Conclusion:

In summary, this code defines a class circle that can calculate the area and perimeter of a circle given its radius. It takes user input for the radius, creates an instance of the circle class, and then calculates and displays the area and perimeter of the circle for that specific radius.

Output:



The screenshot shows a terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, PORTS, and GITLENS. The TERMINAL tab is active. The command prompt shows the user running a Python script. The output displays the area and perimeter of a circle with a radius of 10.

```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\AreaOfCircle.py
Enter radius of circle: 10
Area of circle: 314.16
Perimeter of circle: 62.83
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```


2. To Implement Multiple Inheritance in python.**Code:**

Parent class 1

class Animal:

def __init__(self, name):

self.name = name

def speak(self):

pass

Parent class 2

class Flyable:

def fly(self):

pass

Parent class 3

class Swimmable:

def swim(self):

pass

Child class inheriting from Animal and Flyable

class Bird(Animal, Flyable):

def speak(self):

return f"{self.name} says Tweet!"

def fly(self):

return f"{self.name} is flying."

Child class inheriting from Animal and Swimmable

class Fish(Animal, Swimmable):

def speak(self):

return f"{self.name} says Blub!"

def swim(self):

return f"{self.name} is swimming."

```
# Child class inheriting from Bird and Fish
class Duck(Bird, Fish):
    def __init__(self, name):
        super().__init__(name)

# Create instances of Duck and demonstrate multiple inheritance
donald = Duck("Donald")
print(donald.speak())
print(donald.fly())
print(donald.swim())
```

Conclusion:

This code demonstrates the concept of multiple inheritance in Python, where a class can inherit from multiple parent classes, allowing it to inherit and use the attributes and methods of all its parent classes. In this example, a "Duck" can both fly and swim due to its inheritance from both "Bird" and "Fish" parent classes.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\MultipleInheritance.py
Donald says Tweet!
Donald is flying.
Donald is swimming.
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

3. To Implement a program with the same method name and multiple arguments.**Code:**

```
class MathOperations:
    def add(self, a, b, c=None):
        if c is not None:
            return a + b + c
        else:
            return a + b

    def multiply(self, a, b, c=None):
        if c is not None:
            return a * b * c
        else:
            return a * b

# Create an instance of the MathOperations class
math = MathOperations()

# Call the overloaded methods
result1 = math.add(2, 3)
result2 = math.add(2, 3, 4)
result3 = math.multiply(2, 3)
result4 = math.multiply(2, 3, 4)
print("Result 1:", result1)
print("Result 2:", result2)
print("Result 3:", result3)
print("Result 4:", result4)
```

Conclusion:

The code demonstrates how method overloading can be implemented in Python by defining methods with different numbers of parameters and using conditional statements to handle the variations in the number of arguments. This allows you to perform addition and multiplication operations with flexibility depending on the number of values provided as arguments.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\MethodOverloading.py
Result 1: 5
Result 2: 9
Result 3: 6
Result 4: 24
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

4. To Implement Operator Overloading in python.**Code:**

```
class Vector:
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def __add__(self, other):
```

```
        if isinstance(other, Vector):
```

```
            # Add the x and y components of two vectors and return a new Vector
```

```
            return Vector(self.x + other.x, self.y + other.y)
```

```
        else:
```

```
            raise TypeError("Unsupported operand type for +: " + type(other).__name__)
```

```
    def __str__(self):
```

```
        return f"({self.x}, {self.y})"
```

```
# Create two Vector instances
```

```
v1 = Vector(1, 2)
```

```
v2 = Vector(3, 4)
```


```
# Use the overloaded + operator
```

```
result = v1 + v2
```

```
print(result) # Output: (4, 6)
```

Conclusion:

This code demonstrates how to create a Vector class that can perform vector addition using the + operator. The `__add__` method is overloaded to handle this operation, making it convenient to work with vectors in a more natural and intuitive way.

Output:A screenshot of a terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, PORTS, and GITLENS. The TERMINAL tab is active. It shows a command prompt where the command 'python .\OperatorOverloading.py' has been executed, resulting in the output '(4, 6)'. The prompt is now 'PS F:\Pushkar\MCA\Sem-1\Python Programming>' with a cursor.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\OperatorOverloading.py
(4, 6)
PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

5. Write a program which handles various exceptions in python.**Code:**

try:

```
# Code that may raise exceptions
num1 = int(input("Enter a number: "))
num2 = int(input("Enter another number: "))

result = num1 / num2
print(f"Result: {result}")
```

except ValueError:

```
# Handle the ValueError exception (e.g., if user inputs a non-integer)
print("Please enter valid integer values.")
```

except ZeroDivisionError:

```
# Handle the ZeroDivisionError exception (e.g., if the second number is 0)
print("Cannot divide by zero.")
```

except Exception as e:

```
# Catch all other exceptions
print(f"An error occurred: {e}")
```

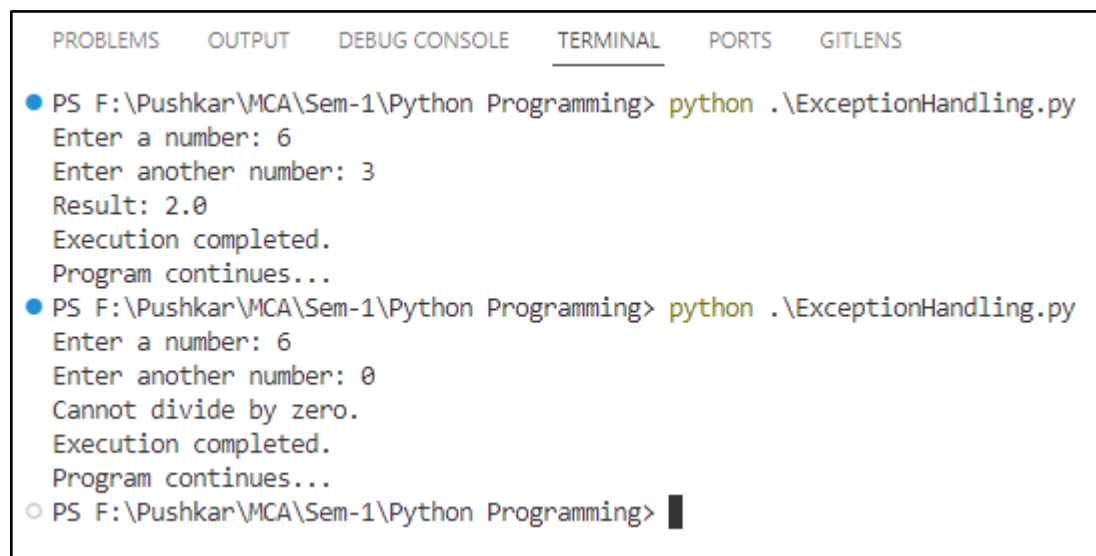
finally:

```
# This block is optional, and it will execute whether there's an exception or not.  
print("Execution completed.")  
# Rest of the program continues here...  
print("Program continues...")
```

Conclusion:

The code showcases how to gracefully handle exceptions and provides error messages for specific error scenarios while ensuring that the program doesn't terminate abruptly due to exceptions. The finally block allows you to execute cleanup or finalization code regardless of whether an exception occurred.

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ExceptionHandling.py  
Enter a number: 6  
Enter another number: 3  
Result: 2.0  
Execution completed.  
Program continues...  
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ExceptionHandling.py  
Enter a number: 6  
Enter another number: 0  
Cannot divide by zero.  
Execution completed.  
Program continues...  
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

Name of Student: Pushkar Sane			
Roll Number: 45		Lab Assignment Number: 6	
Title of Lab Assignment: To implement programs on Data Structures using Python.			
DOP:		DOS:	
CO Mapped: CO3	PO Mapped: PO5, PSO1	Signature:	Marks:

Practical No. 6

Aim: To implement programs on data structure using python.

1. To Create, Traverse, Insert and remove data using Linked List
2. Implementation of stacks
3. Implementation of Queue
4. Implementation of Dequeue

Description:

A data structure is a fundamental concept in computer science and programming. It serves as an organizational framework for efficiently storing, accessing, and manipulating data. Data structures come in various forms, including arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Each data structure has specific characteristics and is chosen based on the problem's requirements. Data structures play a crucial role in optimizing algorithm design and the development of software applications by providing efficient data management and retrieval mechanisms. Understanding data structures is essential for building efficient and organized computer programs.

1. Linked List:

A linked list is a linear data structure consisting of a sequence of elements, each connected to the next element through pointers. Here's a detailed overview of creating, traversing, inserting, and removing data in a linked list:

- **Creation:** To create a linked list, you start with a head node, which is the first element in the list. Each node in the list contains two components: data and a reference (or pointer) to the next node in the sequence.
- **Traversing:** Traversing a linked list involves moving through the list one node at a time. You start at the head and follow the references to each subsequent node until you reach the end (usually when the next reference is null). This allows you to access and display the data in each node.
- **Insertion:** Inserting data in a linked list typically involves adding a new node to the list. To insert data at a specific position, you adjust the references to link the new node correctly within the list.

- **Removal:** Removing data from a linked list requires finding the node to be removed and updating the references to bypass that node. The node can then be deleted.

2. Stack:

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It's commonly used for managing function calls, tracking execution history, and solving problems recursively. Here's a detailed overview of stack operations:

- **Push:** The push operation adds an element to the top of the stack.
- **Pop:** The pop operation removes and returns the top element from the stack.
- **Peek:** The peek operation allows you to view the top element without removing it.
- **Empty:** A check to see if the stack is empty.
- **Size:** Determining the number of elements in the stack.

3. Queue:

A queue is another linear data structure, but it follows the First-In-First-Out (FIFO) principle. It's used in scenarios where tasks should be processed in the order they are received. Here's a detailed overview of queue operations:

- **Enqueue:** The enqueue operation adds an element to the rear (or end) of the queue.
- **Dequeue:** The dequeue operation removes and returns the element at the front of the queue.
- **Front:** Viewing the element at the front of the queue without removing it.
- **Rear:** Viewing the element at the rear of the queue without removing it.
- **Empty:** Checking if the queue is empty.
- **Size:** Determining the number of elements in the queue.

4. Double-Ended Queue (Deque):

A double-ended queue, or deque, is a versatile data structure that allows elements to be added or removed from both ends. It's useful in scenarios where data needs to be efficiently accessed from either end. Here's a detailed overview of deque operations:

- **Enqueue Front:** Adding an element to the front of the deque.
- **Enqueue Rear:** Adding an element to the rear of the deque.
- **Dequeue Front:** Removing and returning the element at the front of the deque.

- **Dequeue Rear:** Removing and returning the element at the rear of the deque.
- **Front:** Viewing the element at the front of the deque without removing it.
- **Rear:** Viewing the element at the rear of the deque without removing it.
- **Empty:** Checking if the deque is empty.
- **Size:** Determining the number of elements in the deque.

These data structures and their associated operations are fundamental building blocks in computer science and software development. Understanding how to create, manipulate, and applying them is essential for solving a wide range of problems efficiently and effectively.

1. To Create, Traverse, Insert and remove data using Linked List.**Code:**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    # Insert at the end of the linked list
    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    # Traverse and print the linked list
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    # Insert data at the beginning of the linked list
    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
```

```
self.head = new_node

# Remove a specific data element from the linked list
def remove(self, data):
    if not self.head:
        return
    if self.head.data == data:
        self.head = self.head.next
        return
    current = self.head
    while current.next:
        if current.next.data == data:
            current.next = current.next.next
            return
        current = current.next

# Example usage
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)

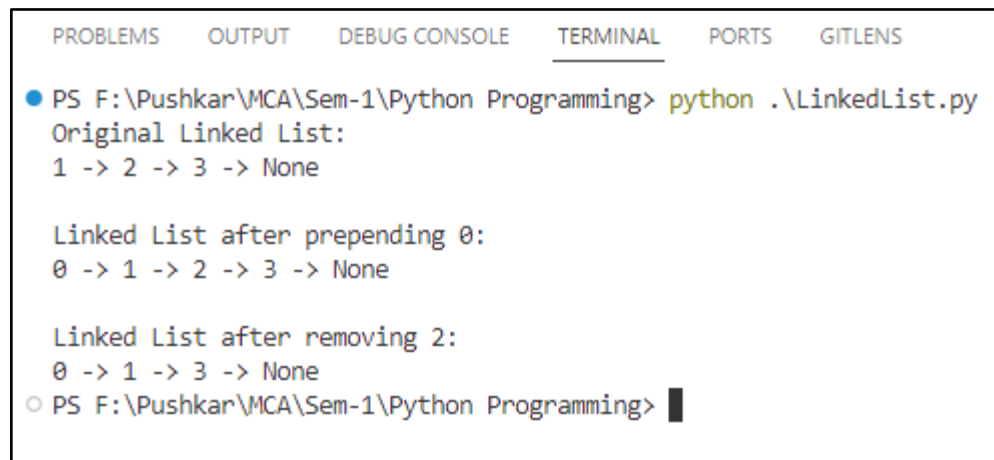
print("Original Linked List:")
linked_list.display()

linked_list.prepend(0)
print("\nLinked List after prepending 0:")
linked_list.display()

linked_list.remove(2)
print("\nLinked List after removing 2:")
linked_list.display()
```

Conclusion:

In conclusion, the Python program presented a basic implementation of a singly linked list, including operations to create, traverse, insert, remove, find the length, search for elements, insert at specific positions, reverse the list, and retrieve elements from the end. This program serves as a practical foundation for understanding and working with linked lists, an essential data structure in computer science and programming.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\LinkedList.py
Original Linked List:
1 -> 2 -> 3 -> None

Linked List after prepending 0:
0 -> 1 -> 2 -> 3 -> None

Linked List after removing 2:
0 -> 1 -> 3 -> None
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

2. Implementation of stack.**Code:**

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
```

```
        else:
            return "Stack is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Stack is empty"

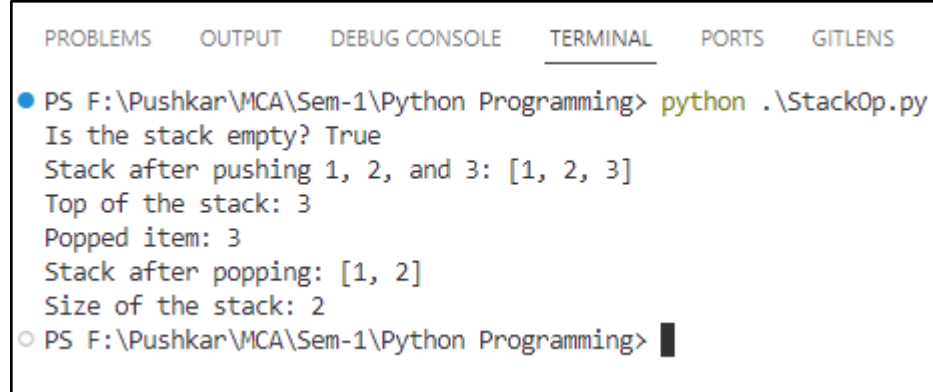
    def size(self):
        return len(self.items)

# Example usage
stack = Stack()
print("Is the stack empty?", stack.is_empty())
stack.push(1)
stack.push(2)
stack.push(3)

print("Stack after pushing 1, 2, and 3:", stack.items)
print("Top of the stack:", stack.peek())
popped_item = stack.pop()
print(f"Popped item: {popped_item}")
print("Stack after popping:", stack.items)
print("Size of the stack:", stack.size())
```

Conclusion:

The Python stack implementation showcased basic stack operations following the Last-In-First-Out (LIFO) principle. Stacks are versatile for tasks like managing function calls and recursion. The code demonstrated pushing, popping, peeking, and checking stack properties, providing fundamental tools for data management and algorithm design.

Output:

The screenshot shows a terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, PORTS, and GITLENS. The TERMINAL tab is active. It displays the execution of a Python script named StackOp.py. The output shows the stack being pushed with values 1, 2, and 3, then the top element (3) is popped, leaving the stack with [1, 2]. The size of the stack is reported as 2. The prompt shows the user is in the directory F:\Pushkar\MCA\Sem-1\Python Programming.

```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\StackOp.py
Is the stack empty? True
Stack after pushing 1, 2, and 3: [1, 2, 3]
Top of the stack: 3
Popped item: 3
Stack after popping: [1, 2]
Size of the stack: 2
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

3. Implementation of Queue.**Code:**

```
class Queue:
```

```
    def __init__(self):
```

```
        self.items = []
```

```
    def is_empty(self):
```

```
        return len(self.items) == 0
```

```
    def enqueue(self, item):
```

```
        self.items.append(item)
```

```
    def dequeue(self):
```

```
        if not self.is_empty():
```

```
            return self.items.pop(0)
```

```
        else:
```

```
            return "Queue is empty"
```

```
    def peek(self):
```

```
        if not self.is_empty():
```

```
            return self.items[0]
```

```
        else:
```

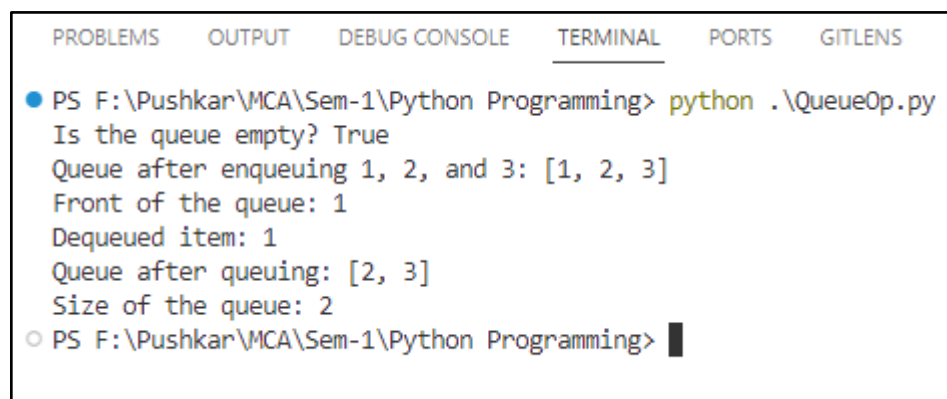
```
            return "Queue is empty"
```

```
def size(self):
    return len(self.items)

# Example usage
queue = Queue()
print("Is the queue empty?", queue.is_empty())
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print("Queue after enqueueing 1, 2, and 3:", queue.items)
print("Front of the queue:", queue.peek())
dequeued_item = queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print("Queue after queuing:", queue.items)
print("Size of the queue:", queue.size())
```

Conclusion:

The Python queue implementation embodies First-In-First-Out (FIFO) behavior, essential for orderly data processing. It showcases basic operations- enqueue, dequeue, peek, and property checks- vital for tasks like task scheduling and data buffering. Queue proficiency is invaluable in software development and computer science.

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\QueueOp.py
Is the queue empty? True
Queue after enqueueing 1, 2, and 3: [1, 2, 3]
Front of the queue: 1
Dequeued item: 1
Queue after queuing: [2, 3]
Size of the queue: 2
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```


4. Implementation of Dequeue.**Code:**

```
class Deque:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def append_right(self, item):
        self.items.append(item)

    def append_left(self, item):
        self.items.insert(0, item)

    def pop_right(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Deque is empty"

    def pop_left(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            return "Deque is empty"

    def peek_right(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Deque is empty"
```

```
def peek_left(self):
    if not self.is_empty():
        return self.items[0]
    else:
        return "Deque is empty"

def size(self):
    return len(self.items)

# Example usage
deque = Deque()
print("Is the deque empty?", deque.is_empty())
deque.append_right(1)
deque.append_right(2)
deque.append_right(3)
print("Deque after appending to the right:", deque.items)
print("Right end of the deque:", deque.peek_right())
deque.append_left(0)
print("Deque after appending to the left:", deque.items)
print("Left end of the deque:", deque.peek_left())
popped_right = deque.pop_right()
print(f"Popped from the right end: {popped_right}")
popped_left = deque.pop_left()
print(f"Popped from the left end: {popped_left}")
print("Deque after popping from both ends:", deque.items)
print("Size of the deque:", deque.size())
```

Conclusion:

In this Python implementation, we introduced a double-ended queue (deque) class with basic operations for efficient element addition and removal from both ends. Deques are versatile data structures useful in various scenarios, such as managing data with distinct priorities and processing elements in an ordered manner. Understanding deque operations is valuable for solving problems in computer science and software development.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\Deque.py
Is the deque empty? True
Deque after appending to the right: [1, 2, 3]
Right end of the deque: 3
Deque after appending to the left: [0, 1, 2, 3]
Left end of the deque: 0
Popped from the right end: 3
Popped from the left end: 0
Deque after popping from both ends: [1, 2]
Size of the deque: 2
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

Name of Student: Pushkar Sane			
Roll Number: 45		Lab Assignment Number: 7	
Title of Lab Assignment: To implement GUI programming and Database Connectivity.			
DOP:		DOS:	
CO Mapped: CO4	PO Mapped: PO3, PO5, PSO1, PSO2	Signature:	Marks:

Practical No. 7

Aim: To implement GUI programming and Database.

1. To Design Login Page.
2. To Design Student Information Form/Library management Form OR Hospital Management Form.
3. Implement Database connectivity For Login Page i.e. Connect Login GUI with Sqlite3.

Description:

Graphical User Interface (GUI) programming in Python allows developers to create user-friendly applications with visual components. Python provides several GUI libraries, with Tkinter being one of the most popular and built into the standard library. Here are some key points on GUI programming with Python:

1. **Choice of Libraries:** Python offers various GUI libraries, including Tkinter, PyQt, PyGTK, and Kivy. Tkinter is a common choice for beginners and is widely used for building desktop applications due to its simplicity.
2. **Creating a GUI Application:** To create a GUI application, you need to create a main application window and add visual elements like buttons, labels, text entry fields, and more.
3. **Event Handling:** GUI programming involves event-driven programming. You define functions (callbacks) that are executed when specific events, like button clicks or mouse movements, occur.
4. **Layout Management:** GUI components need to be organized in a layout. Common layout managers include grid layout, pack layout, and place layout. They help you position components within the window.
5. **User Interaction:** GUIs provide a way for users to interact with the program. This includes input via buttons, text entry, and file dialogs, as well as feedback through labels and message boxes.
6. **Cross-Platform Development:** Python's GUI libraries are often cross-platform, allowing you to develop applications that work on Windows, macOS, and Linux.
7. **Graphics and Styling:** GUI programming allows you to customize the appearance of your application, including fonts, colors, and graphics.

8. **Database Connectivity:** You can connect your GUI application to databases like SQLite, MySQL, or PostgreSQL to store and retrieve data.
9. **Testing and Debugging:** GUI applications require thorough testing, especially for user interactions. Debugging GUIs often involves understanding event flow and component behavior.

Components Used in Designing Login Page and Student Information Form:

1. **Labels:** Labels are used to display static text or descriptions in the GUI. For example, "Username" and "Password" labels in a login page.
2. **Entry Widgets:** Entry widgets allow users to input data. In a login page, these are typically used for username and password input.
3. **Buttons:** Buttons are interactive elements that trigger actions when clicked. A "Login" button is used to initiate the login process.
4. **Message Boxes:** Message boxes are used to display information, alerts, or error messages. They are often used to confirm successful login or display login failures.
5. **Layout Managers:** Grid layout, pack layout, and place layout are used to organize and position the GUI components within the window.

Database Connectivity with SQLite3:

1. **SQLite Database:** SQLite is a lightweight, serverless, and self-contained database engine. It's often used for local data storage in desktop applications.
2. **Database Connection:** To connect to an SQLite database, you create a connection object using `sqlite3.connect('database_name.db')`.
3. **Cursor:** A cursor is used to execute SQL queries and fetch data from the database. You can use the cursor to create tables, insert data, and retrieve data.
4. **SQL Queries:** SQL queries are used to interact with the database. You can perform operations like creating tables, inserting records, selecting data, and updating or deleting records.
5. **Commit and Close:** After executing SQL statements, it's essential to commit changes to the database using `conn.commit()` and then close the connection with `conn.close()` to release resources.

Connecting the GUI with SQLite3 involves using SQL queries to check user credentials, typically during a login operation, as demonstrated in the previous responses. This allows you to verify the user's identity against a database and manage user data.

1. To Design Login Page.**Code:**

```
import tkinter as tk
from tkinter import messagebox
# Function to check login credentials
def login():
    username = entry_username.get()
    password = entry_password.get()

    # Replace this with your own authentication logic
    if username == "Test" and password == "Test123":
        messagebox.showinfo("Login Status", "Login Successful")
    else:
        messagebox.showerror("Login Status", "Login Failed")

# Create the main window
window = tk.Tk()
window.title("Login Page")

# Create labels for username and password
label_username = tk.Label(window, text="Username:")
label_password = tk.Label(window, text="Password:")

# Create entry widgets for user input
entry_username = tk.Entry(window)
entry_password = tk.Entry(window, show="*") # Show '*' for password input

# Create a login button
login_button = tk.Button(window, text="Login", command=login)

# Place widgets on the window using the grid layout
label_username.grid(row=0, column=0)
entry_username.grid(row=0, column=1)
label_password.grid(row=1, column=0)
```

```
entry_password.grid(row=1, column=1)
login_button.grid(row=2, columnspan=2)
```

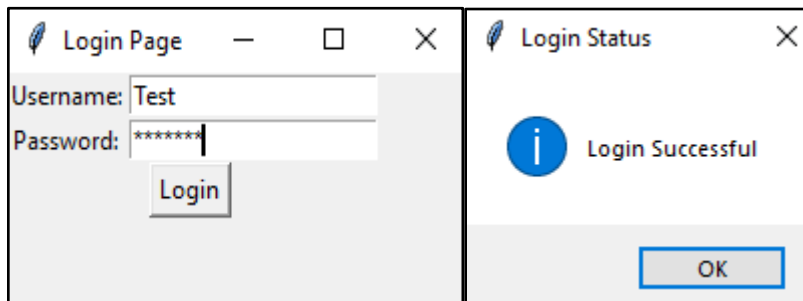
```
# Start the Tkinter main loop
window.mainloop()
```

Conclusion:

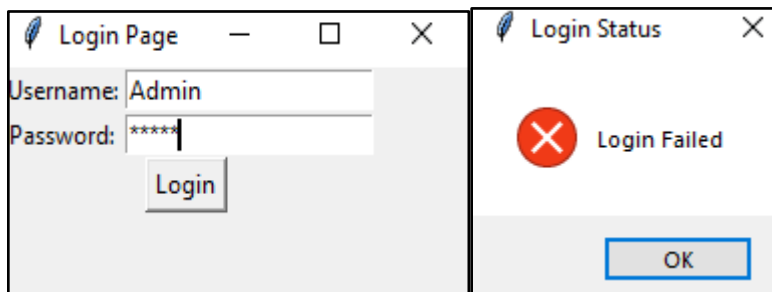
Designing a login page using a GUI in Python is a critical step in creating user-friendly applications. Key points include selecting a GUI library like Tkinter, using visual components (labels, entry fields, buttons), event-driven programming, layout management, user interaction elements, cross-platform compatibility, customization, database connectivity, and the importance of testing and security. A well-designed login page provides a secure and user-friendly entry point for applications.

Output:

Login Successful



Login Failed



2. To Design Student Information Form.**Code:**

```
import tkinter as tk

def save_student_info():
    # Get data from input fields and save to a database or file
    name = entry_name.get()
    roll_number = entry_roll.get()

    # Add data processing logic here
    print(f"Saved Student: {name}, Roll: {roll_number}")

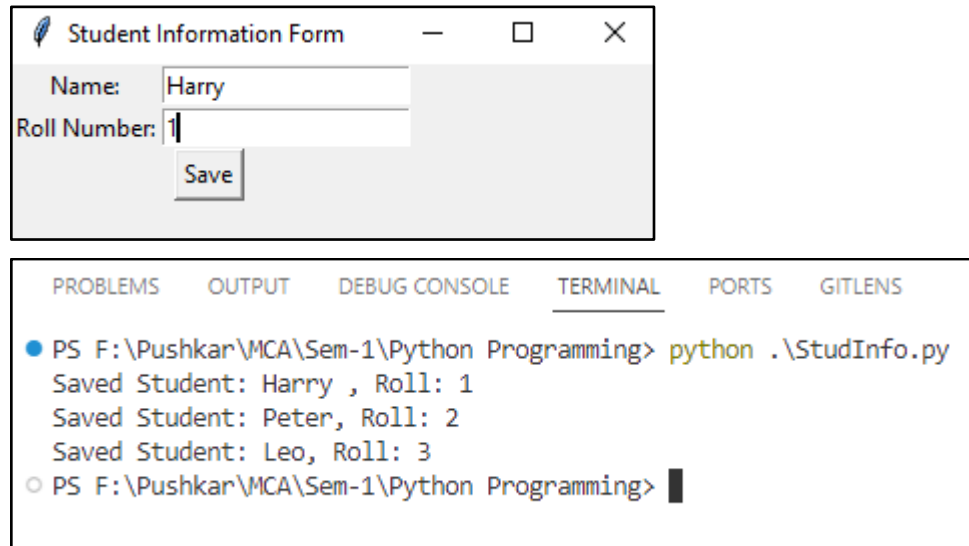
# Create the main window
window = tk.Tk()
window.title("Student Information Form")

# Create labels and entry widgets for student data
label_name = tk.Label(window, text="Name:")
label_roll = tk.Label(window, text="Roll Number:")
entry_name = tk.Entry(window)
entry_roll = tk.Entry(window)
save_button = tk.Button(window, text="Save", command=save_student_info)

# Place widgets on the window using the grid layout
label_name.grid(row=0, column=0)
entry_name.grid(row=0, column=1)
label_roll.grid(row=1, column=0)
entry_roll.grid(row=1, column=1)
save_button.grid(row=2, columnspan=2)
window.mainloop()
```

Conclusion:

Designing a Student Information Form using a GUI in Python allows for efficient data collection and management. It simplifies user input, streamlines data processing, and enhances user experience, making it an essential component for educational institutions and data-driven applications.

Output:**3. Implement Database connectivity For Login Page i.e. Connect Login GUI with Sqlite3.****Code:**

```
import tkinter as tk
import sqlite3
from tkinter import messagebox

# Function to check login credentials
def login():
    username = entry_username.get()
    password = entry_password.get()

    # Connect to the SQLite database
    conn = sqlite3.connect("user_credentials.db")
```

```
cursor = conn.cursor()

cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", ("Test",
"Test123"))
cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)",
("Pushkar", "Admin"))
cursor.execute("""CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    username TEXT NOT NULL,
    password TEXT NOT NULL
)""")

# Check if the user exists in the database
cursor.execute("SELECT * FROM users WHERE username=? AND password=?",
(username, password))
user = cursor.fetchone()
if user:
    messagebox.showinfo("Login Status", "Login Successful")
else:
    messagebox.showerror("Login Status", "Login Failed")

# Close the database connection
conn.close()

# Create the main window
window = tk.Tk()
window.title("Login Page")

# Create labels for username and password
label_username = tk.Label(window, text="Username:")
label_password = tk.Label(window, text="Password:")

# Create entry widgets for user input
entry_username = tk.Entry(window)
```

```
entry_password = tk.Entry(window, show="*") # Show '*' for password input

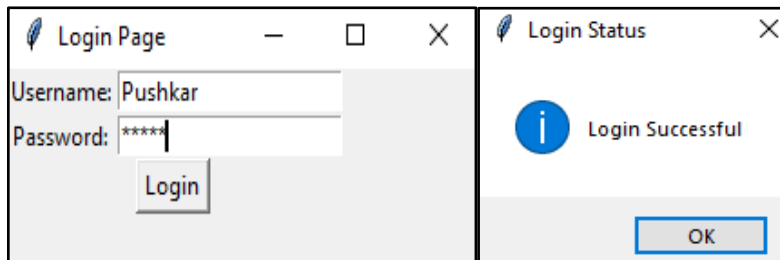
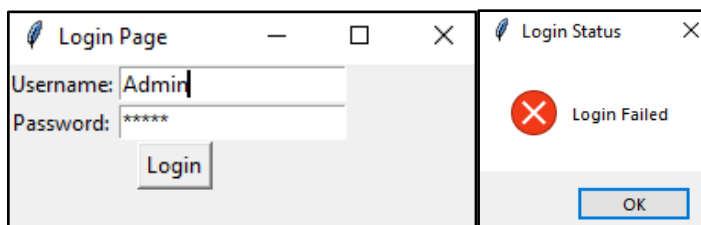
# Create a login button
login_button = tk.Button(window, text="Login", command=login)

# Place widgets on the window using the grid layout
label_username.grid(row=0, column=0)
entry_username.grid(row=0, column=1)
label_password.grid(row=1, column=0)
entry_password.grid(row=1, column=1)
login_button.grid(row=2, columnspan=2)

# Start the Tkinter main loop
window.mainloop()
```

Conclusion:

Connecting a Login GUI with SQLite3 database enables secure user authentication. It enhances data storage and retrieval capabilities, ensuring user credentials are validated effectively, making it a crucial feature for login systems in Python applications.

Output:**User Available In sqlite3 Database****User Not Available In sqlite3 Database**

Name of Student: Pushkar Sane			
Roll Number: 45		Practical Number: 8	
Title of Lab Assignment: To implement Threads in Python.			
DOP:		DOS:	
CO Mapped: CO5	PO Mapped: PO3, PO5, PS01, PS02	Signature:	Marks:

Practical No. 8

Aim: To implement threads in Python.

1. To do design the program for starting the thread in python.
2. Write a program to illustrate the concept of Synchronization.
3. Write a program for creating a multithreaded priority queue.

Description:**Threading:**

Threading is a powerful and essential concept in computer programming that allows for concurrent execution of multiple threads within a single program. Threads are lightweight processes that share the same memory space and can run independently, enabling various functionalities and features. Here are some of the key functionality and features of threading.

Concurrency: Threading provides a way to achieve concurrency in a program. Concurrency allows multiple tasks to make progress and execute independently, leading to improved performance in tasks like I/O-bound operations and responsiveness in applications.

Parallelism: While threading allows for concurrency, it's important to note that due to the Global Interpreter Lock (GIL) in Python, true parallelism in CPU-bound operations is limited. However, threading can still provide benefits when handling I/O-bound tasks, where threads can run in parallel and make better use of multi-core processors.

Creation and Management: Threading in Python is made straightforward through the built-in threading module. This module offers classes and functions for creating, starting, stopping, and managing threads.

Thread Safety: Threads can access shared resources, and this shared access can lead to race conditions and data corruption. Threading provides mechanisms like locks, semaphores, and conditions to ensure thread safety and proper synchronization.

Parallel Task Execution: Threading allows you to break down a larger task into smaller subtasks and execute them in parallel. This is particularly useful in scenarios where you have a set of similar tasks to perform concurrently.

Responsive User Interfaces: Threading is often used in graphical user interfaces (GUIs) to keep the user interface responsive while performing time-consuming operations in the background. This prevents the UI from freezing and improves the user experience.

Resource Sharing: Threads can share resources and data with each other, making it easier to pass information between different parts of your program. However, care must be taken to ensure data integrity through synchronization.

Deadlock Avoidance: Threading provides tools for managing potential deadlocks, a situation where threads wait indefinitely for each other to release resources. Techniques like timeout-based locks and deadlock detection are available for deadlock avoidance.

Distributed Computing: Threading can be used to implement distributed computing and parallel processing. It is often used in scenarios like web scraping, where multiple threads can fetch data from multiple sources simultaneously.

Event Handling: Threads can be used to handle events, such as responding to user input or reacting to changes in data, in a responsive and timely manner.

Load Balancing: Threading can be employed in load balancing scenarios where multiple threads are used to distribute and process tasks among different resources, such as CPU cores.

Task Prioritization: Threading allows you to assign priorities to threads, ensuring that critical tasks are executed before less important ones.

Example:

```
import threading

def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")
```

```
def print_letters():
    for letter in 'abcde':
        print(f"Letter {letter}")

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished.")
```

Synchronization:

Synchronization is the process of coordinating the activities of multiple threads to ensure that they access shared resources or data in a controlled and orderly manner. In a multithreaded environment, synchronization is essential to prevent issues like data corruption, race conditions, and deadlocks. Python offers several mechanisms for synchronization, including locks, semaphores, and condition variables.

Example:

```
import threading

total = 0
lock = threading.Lock()

def increment_total():
    global total
    for _ in range(1000000):
        with lock:
            total += 1
```



```
def decrement_total():
    global total
    for _ in range(1000000):
        with lock:
            total -= 1

thread1 = threading.Thread(target=increment_total)
thread2 = threading.Thread(target=decrement_total)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Final total:", total)
```

Multithreading in Python:

Multithreading in Python is the process of running multiple threads within a Python program. This enables concurrent execution and can significantly improve the performance of certain types of applications. Python's Global Interpreter Lock (GIL) can limit the true parallelism of threads, particularly in CPU-bound operations, but it's still beneficial for I/O-bound tasks.

1. To do design the program for starting the thread in python.**Code:**

```
import threading

# Define a function that will be executed in the thread
def my_thread_function():
    print("Thread is running")

# Create a thread object and pass the function to it
my_thread = threading.Thread(target=my_thread_function)

# Start the thread
my_thread.start()

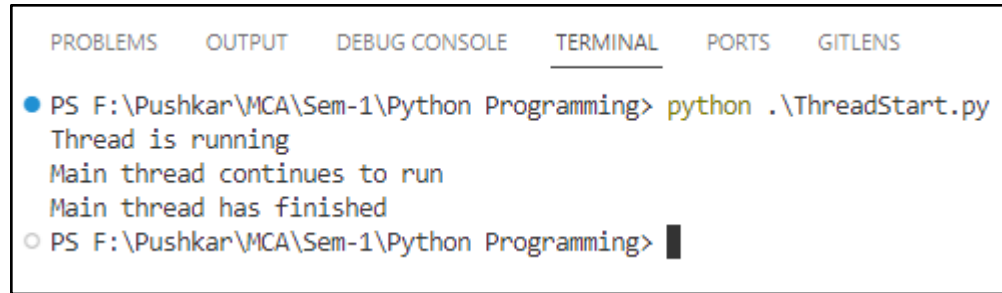
# You can do other work here in the main thread
print("Main thread continues to run")

# Wait for the thread to finish (if needed)
my_thread.join()

# The program continues to execute here
print("Main thread has finished")
```

Conclusion:

The provided code showcases the fundamental principles of threading in Python. A new thread is created, and the `my_thread_function` is executed concurrently with the main thread. Threading allows for parallel execution, enabling the main thread to continue its tasks while the new thread performs its designated function. Proper synchronization with `join()` ensures orderly program termination. This simple example illustrates the power of threading in enhancing program efficiency and responsiveness by executing tasks concurrently.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ThreadStart.py
  Thread is running
  Main thread continues to run
  Main thread has finished
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

2. Write a program to illustrate the concept of Synchronization.**Code:**

```
import threading

# Shared resource (a counter)
counter = 0

# Define a lock for synchronization
counter_lock = threading.Lock()

# Function to increment the counter safely
def increment_counter():
    global counter
    with counter_lock:
        for _ in range(1000000):
            counter += 1

# Create two threads to increment the counter
thread1 = threading.Thread(target=increment_counter)
thread2 = threading.Thread(target=increment_counter)

# Start both threads
thread1.start()
thread2.start()

# Wait for both threads to finish
```

```
thread1.join()
thread2.join()

# Print the final counter value
print("Final counter value:", counter)
```

Conclusion:

The provided code demonstrates the use of threading and synchronization to increment a shared counter safely. Two threads concurrently execute the `increment_counter` function, each adding a million to the counter. A lock (`counter_lock`) is used to ensure exclusive access to the shared resource, preventing race conditions. This example showcases the importance of synchronization in multithreaded applications to maintain data integrity. The final counter value reflects the successful coordination of the threads, emphasizing the power of threading and synchronization in concurrent programming.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ThreadSync.py
Final counter value: 2000000
PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

3. Write a program for creating a multithreaded priority queue.**Code:**

```
import queue
import threading
import time
import random

# Create a priority queue
priority_queue = queue.PriorityQueue()

# Create a flag to signal the retrieval thread to stop
stop_retrieval = threading.Event()
```

```
# Function to insert items into the priority queue
def insert_into_queue():
    for i in range(5):
        item = random.randint(1, 100)
        priority_queue.put((item, f"Item {item}"))
        time.sleep(1)

# Function to retrieve items from the priority queue
def retrieve_from_queue():
    while not stop_retrieval.is_set():
        try:
            item = priority_queue.get(timeout=1)
            print(f"Retrieved: {item[1]}")
            priority_queue.task_done()
        except queue.Empty:
            continue

# Create multiple threads for inserting and retrieving items
insert_thread = threading.Thread(target=insert_into_queue)
retrieve_thread = threading.Thread(target=retrieve_from_queue)

# Start the threads
insert_thread.start()
retrieve_thread.start()

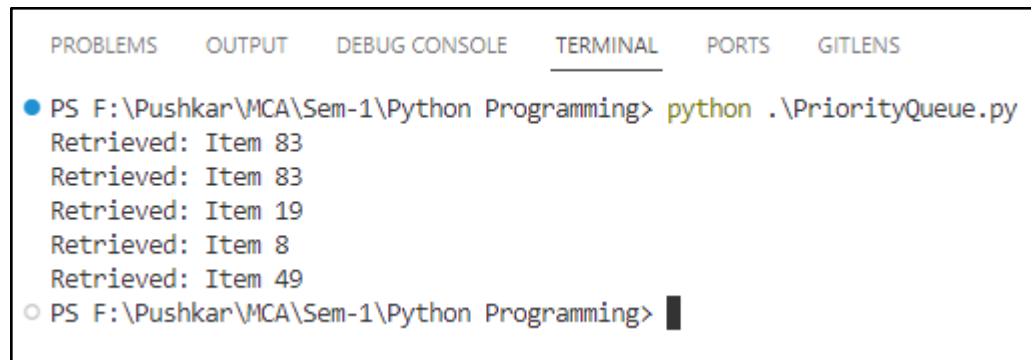
# Wait for the insert_thread to finish
insert_thread.join()

# Set the stop_retrieval flag to signal the retrieve_thread to stop
stop_retrieval.set()

# Wait for the retrieve_thread to finish
retrieve_thread.join()
```

Conclusion:

The provided code demonstrates the use of the `queue.PriorityQueue` and threading in Python for concurrent data insertion and retrieval. An insertion thread adds random items with priorities to the priority queue, while a retrieval thread continuously retrieves and prints items. The code showcases the importance of synchronization and coordination in a multithreaded environment, as the insertion and retrieval threads work together. Additionally, the use of the `threading.Event` allows for graceful termination of the retrieval thread. This example highlights how Python's threading and queue modules can be used to manage shared resources and safely handle concurrent operations.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\PriorityQueue.py
Retrieved: Item 83
Retrieved: Item 83
Retrieved: Item 19
Retrieved: Item 8
Retrieved: Item 49
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

Name of Student: Pushkar Sane			
Roll Number: 45		Practical Number: 9	
Title of Lab Assignment: To implement NumPy library in Python.			
DOP:		DOS:	
CO Mapped: CO6	PO Mapped: PO3, PO5, PS01, PSO2	Signature:	Marks:

Practical No. 9

Aim: To implement NumPy library in Python.

1. Creating an array of objects using array() in NumPy.
2. Creating 2D arrays to implement Matrix Multiplication.
3. Program for Indexing and slicing in NumPy arrays.
4. To implement NumPy - Data Types.

Description:**NumPy:**

NumPy, short for "Numerical Python," is a powerful open-source Python library used for numerical and scientific computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is a fundamental library for data manipulation in various scientific and engineering applications and is an essential building block for many other Python libraries like SciPy, pandas, and scikit-learn.

Key Functionality of NumPy:

1. **Multidimensional Arrays:** NumPy's core data structure is the `numpy.ndarray`, which is an efficient, homogeneous N-dimensional array that can store elements of the same data type. These arrays are highly memory-efficient and enable fast element-wise operations.
2. **Mathematical Operations:** NumPy provides a wide range of mathematical functions for array manipulation. You can perform element-wise operations, vectorized calculations, and linear algebra operations easily.

For example:

```
# Creating arrays
```

```
a = np.array([1, 2, 3])
```

```
b = np.array([4, 5, 6])
```

```
# Element-wise addition
```

```
result = a + b
```

```
print(result) # [5 7 9]
```

```
# Dot product
```

```
dot_product = np.dot(a, b)
```



```
print(dot_product) # 32
```

- 3. Broadcasting:** NumPy allows you to perform operations on arrays of different shapes. It automatically broadcasts the smaller array to match the shape of the larger array, making element-wise operations more flexible.

For example:

```
a = np.array([1, 2, 3])
```

```
b = 2
```

```
result = a + b # Broadcasting b to match the shape of a
```

```
print(result) # [3 4 5]
```

- 4. Array Slicing and Indexing:** NumPy supports powerful slicing and indexing capabilities, making it easy to extract and manipulate subsets of arrays.

For Example:

```
arr = np.array([0, 1, 2, 3, 4, 5])
```

```
# Slicing
```

```
subset = arr[2:5]
```

```
print(subset) # [2 3 4]
```

```
# Indexing
```

```
element = arr[1]
```

```
print(element) # 1
```

- 5. Random Number Generation:** NumPy includes functions for generating random numbers from various probability distributions, which is useful for simulations and statistics.

For Example

```
# Generate an array of random numbers from a uniform distribution
```

```
random_data = np.random.uniform(0, 1, size=(3, 3))
```

```
print(random_data)
```

- 6. Linear Algebra Operations:** NumPy provides functions for linear algebra operations, making it a valuable tool for tasks like matrix multiplication, eigenvalue computation, and solving systems of linear equations.

For Example:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
product = np.dot(A, B)
print(product)
```

- 7. Aggregation and Statistical Functions:** NumPy provides numerous functions for aggregating data and performing statistical calculations, including mean, median, standard deviation, variance, and more.

Example:

```
data = np.array([1, 2, 3, 4, 5])
mean = np.mean(data)
std_dev = np.std(data)
```

- 8. Array Manipulation:** NumPy allows you to reshape, transpose, concatenate, and split arrays easily. These operations are helpful for data preprocessing and manipulation.

Example:

```
array = np.array([[1, 2], [3, 4]])
# Reshape
reshaped = array.reshape((4, 1))
# Transpose
transposed = array.T
# Concatenate
concatenated = np.concatenate((array, array), axis=0)
```

NumPy's efficiency, versatility, and extensive mathematical capabilities make it an essential library for data analysis, machine learning, scientific research, and more in the Python ecosystem.

1. Creating ndarray objects using array() in NumPy.**Code:**

```
import numpy as np

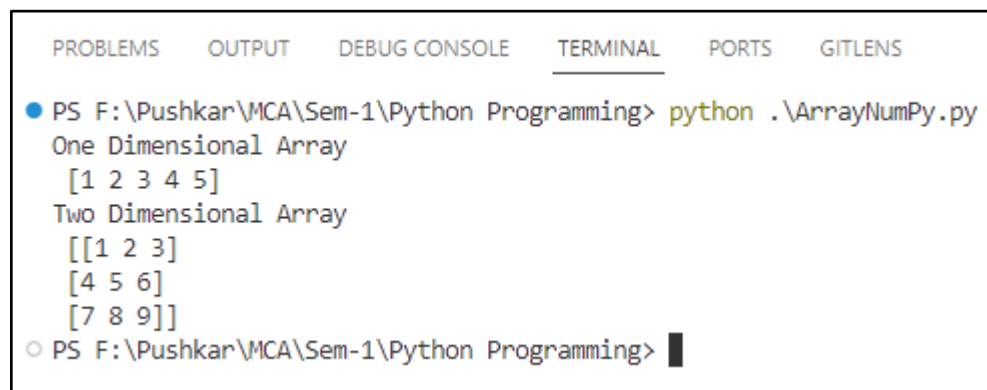
# Creating a 1D array
arr1d = np.array([1, 2, 3, 4, 5])

# Creating a 2D array (matrix)
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print ("One Dimensional Array\n", arr1d)
print ("Two Dimensional Array\n", arr2d)
```

Conclusion:

In this code snippet, we demonstrate the fundamental capabilities of NumPy, a powerful numerical library in Python. We start by creating both a one-dimensional array (arr1d) and a two-dimensional array or matrix (arr2d). NumPy's np.array function allows for efficient storage and manipulation of data, whether in a simple list or a more complex structure like a matrix. This code provides a basic illustration of how NumPy is used to work with arrays, which is just the tip of the iceberg when it comes to the extensive functionality that NumPy offers for numerical and scientific computing in Python.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ArrayNumPy.py
One Dimensional Array
[1 2 3 4 5]
Two Dimensional Array
[[1 2 3]
 [4 5 6]
 [7 8 9]]
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

2. Creating 2D arrays to implement Matrix Multiplication.

Code:

```
import numpy as np

matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])

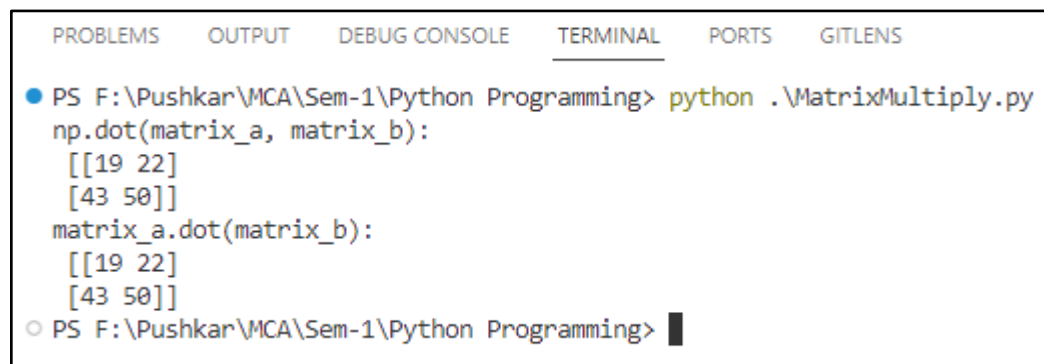
result = np.dot(matrix_a, matrix_b)
print("np.dot(matrix_a, matrix_b): \n", result)

# or
result = matrix_a.dot(matrix_b)
print("matrix_a.dot(matrix_b): \n", result)
```

Conclusion:

In this code snippet, we delve into the power of NumPy for matrix operations. Two matrices, `matrix_a` and `matrix_b`, are created using NumPy arrays. We then utilize the `np.dot` function to perform matrix multiplication, which is a fundamental operation in linear algebra. The result is a new matrix that represents the product of the two original matrices. The code showcases the versatility of NumPy for matrix computations and provides two different ways to achieve the same result: using `np.dot(matrix_a, matrix_b)` or the dot method directly on the array, `matrix_a.dot(matrix_b)`. NumPy simplifies complex mathematical operations, making it an essential tool for scientific and engineering applications in Python.

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\MatrixMultiply.py
np.dot(matrix_a, matrix_b):
[[19 22]
 [43 50]]
matrix_a.dot(matrix_b):
[[19 22]
 [43 50]]
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

3. Program for Indexing and slicing in NumPy arrays.

Code:

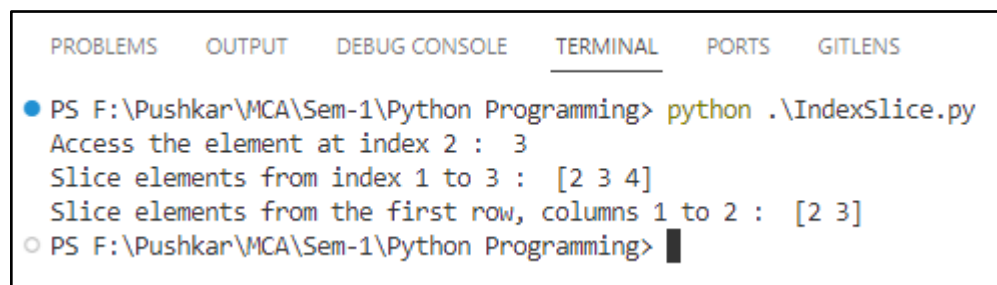
```
import numpy as np
arr1d = np.array([1, 2, 3, 4, 5])
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Indexing
element = arr1d[2] # Access the element at index 2
print("Access the element at index 2 : ", element)

# Slicing
slice1 = arr1d[1:4] # Slice elements from index 1 to 3
print("Slice elements from index 1 to 3 : ", slice1)
slice2 = arr2d[0, 1:3] # Slice elements from the first row, columns 1 to 2
print("Slice elements from the first row, columns 1 to 2 : ", slice2)
```

Conclusion:

In this code snippet, we explore the basics of NumPy for array indexing and slicing. Two arrays, `arr1d` and `arr2d`, are created using NumPy. The code demonstrates how to access individual elements and slices of these arrays. Using index notation, we access the element at index 2 in the one-dimensional array and slice elements from index 1 to 3. Similarly, in the two-dimensional array, we perform a more complex slice operation by specifying both row and column indices. NumPy simplifies data manipulation by providing efficient ways to access and extract specific elements or subsets from arrays, which is fundamental for data analysis and scientific computing in Python.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\IndexSlice.py
Access the element at index 2 : 3
Slice elements from index 1 to 3 : [2 3 4]
Slice elements from the first row, columns 1 to 2 : [2 3]
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

4. To implement NumPy - Data Types.**Code:**

```
import numpy as np

# Specifying data types when creating arrays
arr_int = np.array([1, 2, 3], dtype=np.int32)
arr_float = np.array([1.0, 2.0, 3.0], dtype=np.float64)

# Check the data type of an array
print(arr_int.dtype)
print(arr_float.dtype)

# You can also specify the data type for individual elements
arr_custom = np.array([1, 2, 3], dtype=np.uint16)
print(arr_custom)
```

Conclusion:

This code snippet illustrates how to specify and work with different data types in NumPy arrays. By using the dtype parameter during array creation, you can control the data type of the elements within the array. In this example, we create arrays arr_int and arr_float with explicit data types of int32 and float64, respectively. We then check and print the data types of these arrays using the dtype attribute. Additionally, the code showcases that you can specify the data type for individual elements within an array, as demonstrated with the creation of the arr_custom array with a uint16 data type. NumPy's ability to handle various data types allows for precise control over memory usage and numerical accuracy in scientific and numerical computing tasks.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\DataTypesNumpy.py
int32
float64
[1 2 3]
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

Name of Student: Pushkar Sane			
Roll Number: 45		Practical Number: 10	
Title of Lab Assignment: To implement Pandas library in Python.			
DOP:		DOS:	
CO Mapped: CO6	PO Mapped: PO3, PO5, PS01, PSO2	Signature:	Marks:

Practical No. 10

Aim: To implement Pandas library in Python.

1. Write a Pandas program to create and display a one dimensional array like object containing an array of data using the pandas module.
2. Write a pandas program to convert a dictionary to a Pandas Series.
3. Write a pandas program to create a dataframe from a dictionary and display it. sample data: {'X': [78,85,96,80,86], 'Y': [84,94,89,83,86], 'Z': [86,97,96,72,83]}
4. Write a pandas program to aggregate the two given data frames along rows and assign all data.
5. Write a pandas program to merge two given dataframes with different columns.

Description:

Creating and displaying a one-dimensional array-like object using the Pandas module involves the following key concepts:

1. Pandas Library:

Pandas is a powerful open-source data manipulation and analysis library for Python.

It provides data structures and functions for working with structured data, including Series and DataFrame.

2. Pandas Series:

A Pandas Series is a one-dimensional labeled array that can hold data of any data type.

It is similar to a NumPy array, but with the added feature of having an index associated with each element.

3. Creating a Pandas Series:

To create a Pandas Series, you can use the `pd.Series()` constructor, which accepts a variety of data types, including lists, dictionaries, and NumPy arrays.

The Series constructor allows you to specify the data and optionally customize the index.

4. Displaying a Pandas Series:

You can display the contents of a Pandas Series simply by using the `print()` function or by calling the Series object directly.

When you display a Pandas Series, you'll see both the data values and the associated index (default is an integer index).

Here's a breakdown of the steps in the Pandas program to create and display a one-dimensional array-like object:

1. Import the Pandas library using `import pandas as pd`.
2. Create an array of data (e.g., a list) that you want to convert into a Pandas Series. Use the `pd.Series()` constructor to convert the array of data into a Pandas Series.
3. Optionally, you can customize the index by providing it as an argument when creating the Series. Display the Pandas Series using the `print()` function or by calling the Series object directly.

Converting a dictionary to a Pandas Series using the `pd.Series()` constructor is a straightforward process. This allows you to leverage the powerful features of Pandas, such as data analysis, filtering, and manipulation, with your structured data. The resulting Series is a labeled one-dimensional data structure that can be accessed, indexed, and modified to suit your specific data analysis needs.

A Pandas DataFrame can hold data of various types, including integers, floating-point numbers, strings, or more complex objects.

Each column in the DataFrame can have its own data type, depending on the data provided.

In summary, creating a Pandas DataFrame from a dictionary is a powerful way to work with structured data in Python. It allows you to organize and manipulate data in a tabular format, making it easier to perform data analysis, filtering, and other operations on your data. The resulting DataFrame can be used for various data analysis tasks and is a fundamental building block in data science and data engineering.

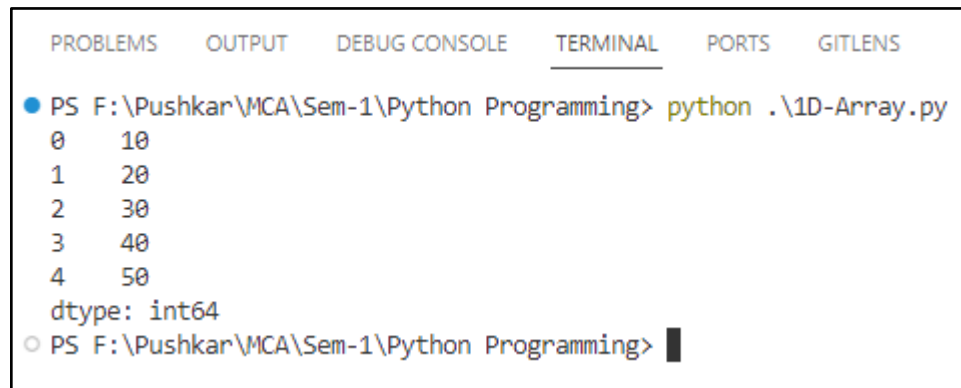
1. **Write a Pandas program to create and display a one dimensional array like object containing an array of data using the pandas module.**

Code:

```
import pandas as pd
data = [10, 20, 30, 40, 50]
my_series = pd.Series(data)
print(my_series)
```

Conclusion:

Here we have successfully created and displayed a one dimensional array-like object containing an array using pandas.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\1D-Array.py
0    10
1    20
2    30
3    40
4    50
dtype: int64
PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

2. **Write a pandas program to convert a dictionary to a Pandas Series.**

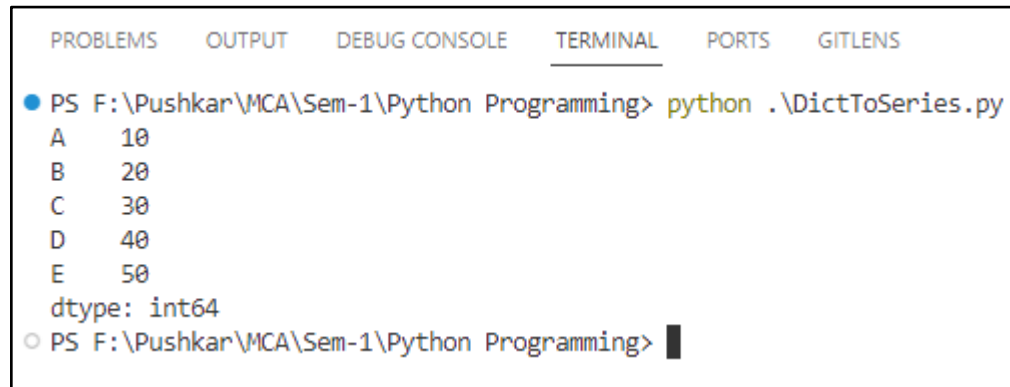
Code:

```
import pandas as pd
# Sample dictionary
data_dict = {'A': 10, 'B': 20, 'C': 30, 'D': 40, 'E': 50}
# Convert the dictionary to a Pandas Series
my_series = pd.Series(data_dict)

# Display the Pandas Series
print(my_series)
```

Conclusion:

Here we have successfully converted a dictionary to a Pandas Series.

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\DictToSeries.py
A    10
B    20
C    30
D    40
E    50
dtype: int64
PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

3. Write a pandas program to create a dataframe from a dictionary and display it.

sample data: {'X': [78,85,96,80,86], 'Y': [84,94,89,83,86], 'Z': [86,97,96,72,83]}.

Code:

```
import pandas as pd
```

```
# Sample data dictionary
```

```
data_dict = {'X': [78, 85, 96, 80, 86], 'Y': [84, 94, 89, 83, 86], 'Z': [86, 97, 96, 72, 83]}
```

```
# Create a Pandas DataFrame from the sample data
```

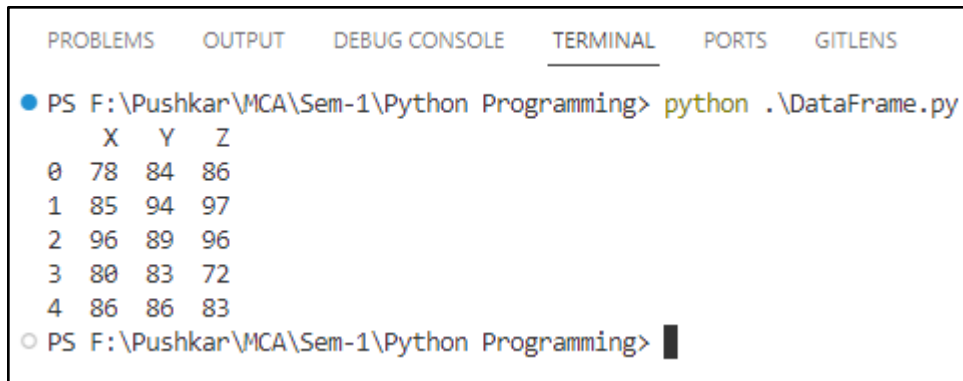
```
df = pd.DataFrame(data_dict)
```

```
# Display the DataFrame
```

```
print(df)
```

Conclusion:

Here we have successfully a dataframe from a dictionary for the given sample data.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\DataFrame.py
      X   Y   Z
0  78  84  86
1  85  94  97
2  96  89  96
3  80  83  72
4  86  86  83
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

4. Write a pandas program to aggregate the two given data frames along rows and assign all data.

Code:

```
import pandas as pd
```

```
# Sample data for two DataFrames
```

```
data1 = {'A': [1, 2, 3], 'B': [4, 5, 6]}
```

```
data2 = {'A': [7, 8, 9], 'B': [10, 11, 12]}
```

```
# Create the first DataFrame
```

```
df1 = pd.DataFrame(data1)
```

```
# Create the second DataFrame
```

```
df2 = pd.DataFrame(data2)
```

```
# Concatenate the two DataFrames along rows and assign all data
```

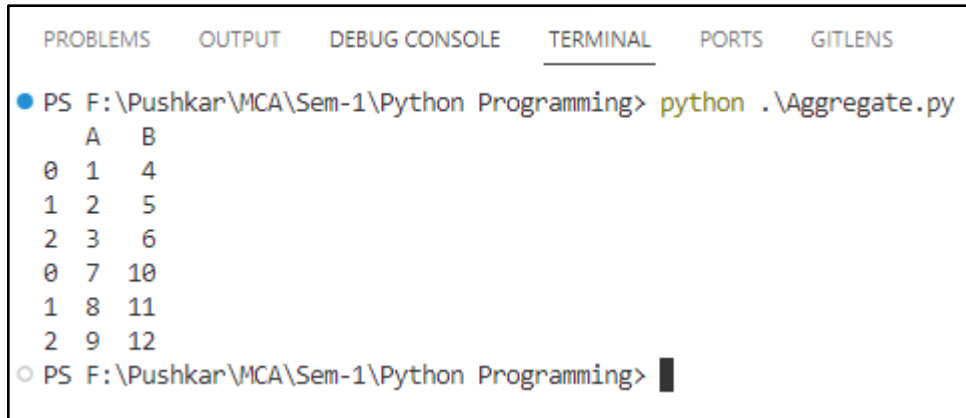
```
result = pd.concat([df1, df2])
```

```
# Display the aggregated DataFrame
```

```
print(result)
```

Conclusion:

Here we have successfully aggregated the two given data frames along rows and assigned all data.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\Aggregate.py
  A  B
0  1  4
1  2  5
2  3  6
0  7 10
1  8 11
2  9 12
PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```

5. Write a pandas program to merge two given dataframes with different columns.**Code:**

```
import pandas as pd

# Sample data for two DataFrames
data1 = {'ID': [1, 2, 3], 'Name': ['Prasad', 'Anish', 'Shreya']}
data2 = {'ID': [2, 3, 4], 'Age': [25, 30, 35]}

# Create the first DataFrame
df1 = pd.DataFrame(data1)

# Create the second DataFrame
df2 = pd.DataFrame(data2)

# Merge the two DataFrames on the 'ID' column
merged_df = pd.merge(df1, df2, on='ID', how='inner')

# Display the merged DataFrame
print(merged_df)
```

Conclusion:

Here we have successfully merged two given dataframes with different columns.

Output:

The screenshot shows a terminal window with a menu bar at the top containing 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected and underlined), 'PORTS', and 'GITLENS'. Below the menu bar, there are two command prompts. The first prompt is 'PS F:\Pushkar\MCA\Sem-1\Python Programming>' followed by the command 'python .\MergeDataFrame.py'. The output of this command is a table with three columns: 'ID', 'Name', and 'Age'. The table contains two rows of data: the first row has '0' for ID, '2' for Name, and 'Anish' for Age; the second row has '1' for ID, '3' for Name, and 'Shreya' for Age. The second command prompt is 'PS F:\Pushkar\MCA\Sem-1\Python Programming>' followed by a cursor.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

● PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\MergeDataFrame.py
  ID    Name  Age
0    2  Anish  25
1    3  Shreya  30
○ PS F:\Pushkar\MCA\Sem-1\Python Programming> █
```